



Qattous, Hazem Kathem (2011) *Constraint specification by example in a meta-CASE tool*. PhD thesis.

<http://theses.gla.ac.uk/2807/>

Copyright and moral rights for this thesis are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the Author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the Author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given



University of Glasgow | School of
Computing Science

Constraint Specification by Example in a Meta-CASE Tool

by

Hazem Kathem Qattous

**Submitted in fulfilment of the requirements for
the degree of Doctor of Philosophy**

School of Computing Science

Faculty of Information and Mathematical Sciences

University of Glasgow

August 2011

Abstract

Meta-CASE tools offer the ability to specialise and customise diagram-based software modelling editors. Constraints play a major role in these specialisation and customisation tasks. However, constraint definition is complicated. This thesis addresses the problem of constraint specification complexity in meta-CASE tools. Constraint Specification by Example (CSBE), a novel variant of Programming by Example, is proposed as a technique that can simplify and facilitate constraint specification in meta-CASE tools. CSBE involves a user presenting visual examples of diagrams to the tool which engages in a synergistic interaction with the user, based on system inference and additional user input, to arrive at the user's intended constraint.

A prototype meta-CASE tool has been developed that incorporates CSBE. This prototype was used to perform several empirical studies to investigate the feasibility and potential advantages of CSBE. An empirical study was conducted to evaluate the performance in terms of effectiveness, efficiency and user satisfaction of CSBE compared to a typical form-filling technique. Results showed that users using CSBE correctly specified significantly more constraints and required less time to accomplish the task. Users reported higher satisfaction when using CSBE. A second empirical online study has been conducted with the aim of discovering the preference of participants for positive or negative natural language polarity when expressing constraints. Results showed that subjects preferred positive constraint expression over negative expression. A third empirical study aimed to discover the effect of example polarity (negative vs. positive) on the performance of CSBE. A multi-polarity tool offering both positive and negative examples scored significantly higher correctness in a significantly shorter time to accomplish the task with a significantly higher user satisfaction compared to a tool offering only one example polarity. A fourth empirical study examined user-based addition of new example types and inference rules into the CSBE technique. Results demonstrated that users are able to add example types and that performance is improved when they do so.

Overall, CSBE has been shown to be feasible and to offer potential advantages compared to other commonly-used constraint specification techniques.

Table of Contents

Chapter 1 Introduction	1
1.1 Introduction	2
1.2 Research Problem and its Context	2
1.3 The Proposed Solution	7
1.4 Aims and Thesis Statement	8
1.5 Approach	10
1.6 Chapter Summary	12
1.7 Dissertation Roadmap	13
Chapter 2 Background	18
2.1 Introduction	19
2.2 Computer Aided Software Engineering (CASE) Tools	19
2.3 CASE Tools Limitations	22
2.4 Domain Specific Languages and Tools	23
2.5 Meta-CASE Tools	27
2.6 Meta-modelling	30
2.7 Constraints (Related to Study One, Chapter 5)	38
2.7.1 Importance of Constraints in Meta-modelling	38
2.7.2 Constraint Definition	41
2.7.2.1 Formal Constraint Language	42
2.7.2.2 Text-based Constraint Specification Language	44
2.7.2.3 Visual Programming Language	46
2.7.2.4 Spreadsheets	48
2.7.2.5 Form-Filling	49
2.7.3 Summary of Meta-CASE tools	51
2.8 Programming by Example (Related to Study One, Chapter 5)	52
2.8.1 Programming by Example Contexts and Tools	54
2.9 Example Polarity (Related to Studies 2 & 3, Chapter 6)	58
2.10 Rules Augmentation and Learning (Related to Study Four, Chapter 7)	61
2.11 Conclusion	67
Chapter 3 Diagram Editor Constraint System (DECS)	68
3.1 Introduction	69
3.2 Diagram Editor Constraints System (DECS) Structure and History	69

3.2.1	Overview	69
3.2.2	Development History	72
3.2.3	DECS Limitations and Enhancements in this Research	73
3.3	The Meta-Modelling in DECS	74
3.4	The Constraint Language in DECS	79
3.5	Constraint Specification	85
3.5.1	Constraint Specification Using the Form-Filling Technique	86
3.6	Why DECS?	89
3.7	Conclusion	91
Chapter 4 Constraint Specification by Example (CSBE)		93
4.1	Introduction	94
4.2	Constraint Specification by Example.....	94
4.3	Visualisation-Oriented Constraint Specification	95
4.4	CSBE Model	95
4.5	CSBE Design and Implementation.....	98
4.5.1	Positive and Negative Examples	103
4.5.2	Inference Over States and Actions.....	106
4.5.3	Inference Engine Transparency.....	112
4.6	The Second Example: Synergistic Approach and Example Remodelling (Visual Generalisation).....	115
4.7	Inference Engine Rules.....	118
4.8	Inference Engine Augmentation and Learning	120
4.8.1	Generalisation	127
4.8.2	Some Complicated Scenarios.....	132
4.9	Conclusion.....	136
Chapter 5 : Research Methods, Population Definition and General Threats to Validity		139
5.1	Introduction	140
5.2	Research Methods	140
5.2.1	Experimental Design Principles.....	140
5.2.2	Pilot Studies	140
5.2.3	Screen capturing and recording.....	141
5.2.4	Questionnaires.....	143
5.2.5	Interview	144
5.3	Sample Selection.....	145

5.3.1	Population Definition	145
5.3.2	Deviation from the Well-Defined Population	146
5.3.3	Drawing Samples from the Population	147
5.4	Threats to Validity	148
5.5	Conclusion	150
Chapter 6 Empirical Studies		151
6.1	CSBE vs Form-Filling Technique (Study One)	152
6.1.1	Introduction	152
6.1.2	Experimental Methodology	153
6.1.2.1	Aim and Hypothesis	153
6.1.2.2	The Hypothesis.....	153
6.1.2.3	Collection and Tasks.....	153
6.1.2.4	Participants	155
6.1.2.5	Experimental Design	155
6.1.3	Results.....	156
6.1.3.1	Correctness	156
6.1.3.1.1	State Transition Diagram	157
6.1.3.1.2	Use Case Diagram	157
6.1.3.2	Time.....	158
6.1.3.2.1	State Transition Diagram	159
6.1.3.2.2	Use Case Diagram	161
6.1.3.3	User Perception.....	163
6.1.4	Discussion.....	165
6.1.5	Threats to Validity	168
6.1.6	Related Work	169
6.1.7	Conclusion.....	170
6.2	Constraint Polarities in Natural Language (Study Two)	171
6.2.1	Introduction	171
6.2.2	Constraint Polarity Survey.....	171
6.2.2.1	Aim	171
6.2.2.2	Experimental Design	171
6.2.2.3	Results	172
6.2.3	Discussion.....	173
6.2.4	Threats to Validity	174

6.2.5	Conclusion	174
6.3	Example Polarities (STUDY THREE)	175
6.3.1	Introduction	175
6.3.2	Positive and Negative Examples in DECS	176
6.3.3	Inference Over State and Action	177
6.3.4	Effect of Example Polarity	179
6.3.4.1	Aim	179
6.3.4.2	Experimental Units, Materials, and Tasks	179
6.3.4.3	Variables and Hypothesis	185
6.3.4.4	The Hypothesis:	185
6.3.4.5	Experimental Design and Execution	185
6.3.4.6	Results	187
6.3.4.6.1	Correctness	187
6.3.4.6.1.1	Correctness for each participant	188
6.3.4.6.1.2	Correctness for each constraint	188
6.3.4.6.2	Time	190
6.3.4.6.2.1	Time spent by each user to accomplish the task	190
6.3.4.6.2.2	Time spent to define each constraint by all the participants	191
6.3.4.6.3	User Perception	192
6.3.5	Discussion	195
6.3.6	Threats to Validity	197
6.3.7	Related Work	198
6.4	Adding and Customising Rules (STUDY FOUR)	200
6.4.1	Introduction	200
6.4.2	Proposed Solution	202
6.4.3	The Feasibility and Desirability of Adding Rule and Customisation Feature	204
6.4.3.1	Aim, Variables and Hypothesis	204
6.4.3.2	The independent variables:	204
6.4.3.3	The dependent variables	205
6.4.3.4	The Hypothesis:	205
6.4.3.5	Data Collection and Tasks	206
6.4.3.6	Experimental Design and Execution	210
6.4.4	Results	213
6.4.4.1	Correctness	213

6.4.4.1.1	Correctness for each participant	213
6.4.4.1.2	Correctness for each constraint.....	215
6.4.4.2	Number of constraints required to be taught (added)	216
6.4.4.2.1	Number of constraints required to be taught for each participant	216
6.4.4.2.2	Number of constraints required to be taught for each constraint.....	217
6.4.4.3	Time.....	220
6.4.4.3.1	Sum of time required by each participant to complete the tasks	220
6.4.4.3.2	Sum of time required by all the participants for each constraint	221
6.4.4.4	User Perception.....	223
6.4.4.5	Discussion.....	225
6.4.5	Threats to Validity	228
6.4.6	Related work	229
Chapter 7	Summary, Conclusions and Future Work.....	231
7.1	Introduction	232
7.2	Research Contributions and Achievements.....	232
7.3	Thesis Summary	233
7.4	Conclusions and Recommendations	236
7.5	Future work.....	241
7.5.1	More Diagram types to be Involved in the Research.....	241
7.5.2	Enhancing the Constraint Language	242
7.5.3	Comparing CSBE with other constraint specification techniques.....	243
7.5.4	Ranking the Inferences	243
7.5.5	Recommendation System Depending on Previous Specifications.....	244
7.5.6	Enhance Rule Addition by Selecting the Required Collected Features.....	245
7.5.7	Visual Language.....	246
7.5.8	Constraint Conflict Checker	246
Bibliography	248
Appendix A	Constraint XML Files	262
A.1	Vertex Constraint XML File Template	263
A.2	Edge Constraint XML File Template	265
Appendix B	Diagram Constraint Lists	269
B.1	State Transition Diagram Main Constraint List	270
B.2	State Transition Diagram Constraint Categories.....	271
B.3	Use Case Diagram Main Constraint List	272

B.4	Use Case Diagram Constraint Categories.....	273
Appendix C	Study One.....	274
C.1	Study One Constraint Lists	275
C.1.1	Constraint List for User 1, <i>State Transition Diagram</i>	275
C.1.2	Constraint List for User 1, <i>Use Case Diagram</i>	275
C.2	Study One Post-Experiment Questionnaire	276
C.3	Study One Exit Questionnaire / Interview	279
Appendix D	Study Two.....	284
D.1	Study Two Questionnaire:.....	285
Appendix E	Study Three	288
E.1	Study Three Constraint Lists	289
E.1.1	Constraint List 1	289
E.1.2	Constraint List 2	289
E.2	Questions Per Constraint	290
E.3	Study Three Post-Experiment Questionnaire	291
E.4	Study Three Exit Questionnaire / Interview.....	293
Appendix F	Study Four	296
F.1	Inference Features	297
F.1.1	State Inference Features.....	297
F.1.2	Action Inference Features.....	297
F.1.3	Visual Generalisation Inference Features	298
F.1.4	Rule Generalisation Features.....	298
F.2	Study Four Constraint Lists	298
F.2.1	Constraint List 1	298
F.2.2	Constraint List 2	298
F.2.3	Constraint List 3	299
F.2.4	Constraint List 4	299
F.3	Study Four Post-Experiment Questionnaire	301
F.4	Study Four Exit Questionnaire / Interview.....	303

Table of Figures

Figure 1-1: Visualisation of a constraint that cannot be expressed using the XML-based language used in DECS for the purpose of this research.....	11
Figure 1-2: The thesis chapters and roadmap.	15
Figure 2-1: A meaningless diagram constructed on Gliffy that mixes the Use Case Diagram with Class Diagram vocabularies.....	21
Figure 2-2: Multi-view mobile application using DSM language. Adopted from (MetaCase, 2009).	25
Figure 2-3: Specification of insurance product using financial DSM language. Adopted from (MetaCase, 2009).....	25
Figure 2-4: DSM language for designing car infotainment systems. Adopted from (MetaCase, 2009).	26
Figure 2-5: DSM language for modelling the layout of the railway track. Adopted from (MetaCase, 2009).	26
Figure 2-6: a) A domain specific diagram editor for modelling signal flow on a chip design environment generated using the meta-CASE tool GME. Adopted from (Systems, 2011). b) A domain specific diagram editor for modelling finite state machine. Adopted from (Ledeczi, Maroti, & Volgyesi, 2004).....	28
Figure 2-7: Using MetaEdit+ meta-CASE tool for developing Family Tree Language Editor. a) the required target modelling language in use. b) the meta-modelling concepts over the required language. c) the graphical meta-model of the required language. Adopted from (MetaCase, 2009).	35
Figure 2-8: A graphical description of a hypernode. Ellipses are hypernodes, solid lines are hypernode edges and the dotted lines indicate hypernode nesting. Adopted from (Scott, 1997).	36
Figure 2-9: Meta-model of the finite state machine domain specific language using class diagram as a meta-modelling language in GME meta-CASE tool. Adopted from (Systems, 2011).....	37
Figure 2-10: A meta-model in GME meta-CASE tool using class diagram and a constraint has been annotated to one of the classes. The “Equation” attribute of the constraint shows the OCL expression “self.parts()->size > 0”. Adopted from (GME, 2005).	43

Figure 2-11 Constraint specification using Java programming language in Pounamu. Adopted from (Zhu, Grundy, & Hosking, 2004).	44
Figure 2-12: a) Using Kaitiaki for specifying layout constraint. b) reusable sub-query which is called by the query “getSubPages” in (a). Adopted from (Liu, Hosking, & Grundy, 2007a).	47
OCL expression.....	49
Figure 2-13: Spreadsheet-like GUI (adapted from Liu, Hosking, & Grundy(2007b))	49
Figure 2-14: Form-filling technique in MetaEdit+. a) cardinality constraint specification. b) connectivity constraint specification. Adopted from (MetaCase, 2009).	50
Figure 2-15: Specifying a cardinality constraint using form-filling technique in MetaBuilder meta-CASE tool. Adopted from (Gong, Scott, & Offen, 1997).	50
Table 2-2: Programming By Example different domains of application.....	63
Figure 2-16: Diagram editor specification and use in ISI. Adopted from (Goldman & Balzer, 1999).	64
Figure 2-17: Specifying the GUI of a diagram editor for modelling satellite communications. Adopted from (Goldman & Balzer, 1999).	65
Figure 2-18: The generated specified diagram editor for modelling Satellite Communications. Adopted from (Goldman & Balzer, 1999).	66
Figure 3-1: Interaction between Eclipse and the DECS plug-in. Adopted from (Inglis, 2005).	70
Figure 3-2: DECS Initial Structure and Components Interaction.	71
Figure 3-3: DECS Use by different types of users.....	71
Figure 3-4: a) Select the required element (new vertex type). b) Enter the vertex type name. c) Define the presentation properties for the new define type.....	76
Figure 3-5: Select the vertices participating as elements at State Transition Diagram.	77
Figure 3-6: a) The user creates a new diagram of the specified diagram types. b) The user gives a name for the diagram and selects the required diagram type.....	78
Figure 3-7: Generated State Transition Diagram editor.....	79
Figure 3-8: A soft constraint asking the user to add an End State.	82

Figure 3-9: Edge constraint file referring to End State as a source connection and to Start State as a target connection constraint files.....	83
Figure 3-10: Edge constraint file referring two times to the same vertex constraint file.	84
Figure 3-11: a) Transition edge constraint form (wizard) that defines two new constraints as references. b) End State vertex constraint form (tabbed-frame) to define referenced source connection constraint from the edge constraint. c) Start State vertex constraint form (tabbed-frame) to define referenced source connection constraint from the edge constraint.	88
Figure 3-12: The error message “It is not allowed to have more than one start state in the same diagram” indicating violation of the constraint when the user tries to add the second start state.	89
Figure 4-1: Synergistic interaction model in DECS. Interaction loop is shown in bold.	97
Figure 4-2: DECS Structure and the components interaction after the inference manager component is added.....	98
Figure 4-3: DECS is used by different types of users.....	99
Figure 4-4: a) Negative example to express the constraint “ <i>it is not allowed to connect two vertices of type Actor using an edge of type Association</i> ”. b) Inferred constraints list.....	101
Figure 4-5: Generated diagram editor working. The error message “It is not allowed to have the structure (Actor is connected to Actor) using Association” indicating violation of the constraint when the user tried to connect two Actors using Association edge.	102
Figure 4-6: Positive interpretation (inference) the introduced example.	103
Figure 4-7: Negative inference for the same example introduced in (Figure 4-6).....	104
Figure 4-8: Positive interpretation of the same example in Figure 4-3.....	105
Figure 4-9: a) The user introduces a vertex (Actor) as an initial state of the example and the system shows the negative inference in the negative version of DECS. b) The user deletes the vertex to express the required constraints negatively using the state-action feature.....	107
Figure 4-10: a) Initial state for an example. b) Expressing the required constraint negatively using state-action feature by deleting one of the Include type edges.	108

Figure 4-11: Positive interpretation (inference) the introduced example.	109
Figure 4-12: a) Initial state for an example. b) Expressing the required constraint positively using state-action feature by deleting one of the Actor vertices.	110
Figure 4-13: a) Initial state for an example. b) Expressing the required constraint positively using state-action feature by deleting the Association edge.	111
Figure 4-14: The user is watching the inference engine working and reading the inferences while building the required example.	113
Figure 4-15: The user is watching the inference engine working and reading the inferences while building the required example.	114
Figure 4-16: Visual generalisation by remodelling in DECS. (a) examples by the user, (b) example remodelling.	115
Figure 4-17: Visual generalisation. (a) examples by the user, (b) first remodelling, (c) second remodelling.	117
Figure 4-18: the system could not infer the required constraint and the user is adding a new rule.	121
Figure 4-19: Positive example expressing the constraint “ <i>It is not allowed to have more than 1 Start State in a state transition diagram</i> ”.....	122
Figure 4-20: The system provides the user with a form to define a graph constraint.	123
Figure 4-21: The user enters some properties as a first step.	124
Figure 4-22: a) Continue specifying the constraint UBN 1. b) Specification of the Start State as a separate constraint that is referenced from UBN 1.	125
Figure 4-23: The system learned how to infer the required constraint, from the preferred example and preferred polarity.....	126
Figure 4-24: Positive example expressing the constraint “ <i>It is not allowed to have more than 1 End State in a state transition diagram</i> ”.	127
Figure 4-25: The ‘added rules’ XML file	128
Figure 4-26: Generalisation to infer from previously learned example.....	131
Figure 4-27: Example expressing the constraint “ <i>It is not allowed to have more than 2 Start State vertices in a state transition diagram</i> ”.	133

Figure 4-28: Example expressing the constraint “ <i>It is not allowed to have more than 3 end state vertices in a state transition diagram</i> ”.	133
Figure 4-29: Example expressing the constraint “ <i>There must be at least one edge of type Transition connecting green start state vertex (as a source) with red non-terminal state vertex (as a target) in a state transition diagram</i> ”.	134
Figure 4-30: Example expressing the constraint “ <i>There must be at least one edge of type Transition connecting yellow non-terminal state vertex (as a source) with blue end state vertex (as a target) in a state transition diagram</i> ”.	135
Figure 4-31: Negative example expressing the constraint “ <i>There must be at least one edge of type Transition connecting green start state vertex (as a source) with red non-terminal state vertex (as a target) in a state transition diagram</i> ”.	135
Figure 4-32: Negative example expressing the constraint “ <i>There must be at least one edge of type Transition connecting yellow non-terminal state vertex (as a source) with blue end state vertex (as a target) in a state transition diagram</i> ”.	136
Figure 4-33: a) A complicated example in DECS. b) The inference from the example.	138
Figure 6-1: Number of constraints defined correctly using wizard and CSBE by each user in the STD.	157
Figure 6-2: Number of constraints defined correctly using wizard and CSBE by each user in the UCD.	158
Figure 6-3: Task completion time in minutes for Wizard and CSBE by each user in the STD.	160
Figure 6-4: Kaplan-Meier survival curve of STD.	161
Figure 6-5: Task completion time in minutes for the Wizard and CSBE by each user in the UCD.	162
Figure 6-6: Kaplan-Meier survival curve of UCD.	163
Figure 6-7: Percentage of participants agree that the constraint expression in natural language is easier to be positive, negative or both are equal.	173
Figure 6-8: Positive interpretation (inference) the introduced example.	177
Figure 6-9: Negative inference for the same example introduced in (Figure 6-8).	177
Figure 6-10: a) The user introduces a vertex (Actor) as an initial state of the example and the system shows the negative inference in the negative version of DECS. b) The	

user deletes the vertex to express the required constraints negatively using the state-action feature.....	178
Figure 6-11: Number of constraints defined correctly using Negative-Action and Negative-Positive tools.	188
Figure 6-12: Number of participants correctly defined each constraint using NA and NP tools.....	189
Figure 6-13: Sum of time required by each participant to complete the task in minutes using NA and NP tools.....	190
Figure 6-14: Sum of time required by all participants to define each constraint alone.	191
Figure 6-15: Adding rules process within CSBE.....	203
Figure 6-16: Correctness for each participant in task one and task two out of 6 constraints.	214
Figure 6-17: Correctness for each constraint for task one and task two out of 16 participants.	215
Figure 6-18: Number of constraints required to be taught using “adding rule feature out of 6 constraints.....	217
Figure 6-19: Number of times the adding rule feature is used for each constraint out of 16 in task1 and task2.....	218
Figure 6-20: The total time required by each participant out of 16 to accomplish the task.	221
Figure 6-21: Total time required by all the participants to accomplish the specification task for each constraint.	222

Table of Tables

Table 1-1: Contributions distributed over thesis chapters	16
Table 1-2: Achievements distributed over thesis chapters.....	17
Table 2-1: Meta-CASE tools classification based on the constraint specification technique.	52
Table 6-1: User perception of wizard and CSBE techniques in STD and UCD. (higher = better (higher user satisfaction), bold and underline = significant difference)	164
Table 6-2: User perception of wizard and CSBE techniques in STD and UCD. (lower = better (higher user satisfaction), bold and underline = significant difference)	165
Table 6-3: Natural language constraint expression preference study summary	172
Table 6-4: Possible implementation alternatives for example polarities.	179
Table 6-5: Design of the experiment and user first assignments to different counterbalanced conditions.....	184
Table 6-6: User perception of the NA and the NP tools in post-constraint questionnaire. (shaded = better (higher user satisfaction), bold and underlined = significant difference)	193
Table 6-7: User perception of the NA and the NP tools in post-task questionnaire. (higher = better (higher user satisfaction), bold and underlined = significant difference)	194
Table 6-8: User perception of the NA and the NP tools in post-task questionnaire. (lower = better (higher user satisfaction), bold and underlined = significant difference)	195
Table 6-9: Justification of the reasons for using each constraint in one of the lists used in the study.	208
Table 6-10: Constraints used in the study and the ability of the system to infer them before using the adding rule feature (teaching the system).....	209
Table 6-11: Assignment of lists to users.....	211
Table 6-12: The constraints that required the use of rule addition feature in task one (shaded = added).	219

Table 6-13: The constraints that required the use of rule addition feature in task two (shaded = added).	220
Table 6-14: Average answers of the post-task questionnaire. (shaded = better (higher user satisfaction), bold and underline = significant difference).....	223
Table 6-15: Post-experiment questionnaire.	224
Table 7-1: Contributions distributed over thesis chapters	232
Table 7-2: Achievements distributed over thesis chapters.....	233

Acknowledgment

First of all I would like to thank my first supervisor Philip Gray. He was the great manager who guided me to the right path throughout this research. Philip Gray with his management, experience, endless help and support, participated significantly in this work completion. I will never forget the days I met him for an hour every time I went for coffee. I called these unplanned meetings “coffee meetings” and I knew I was taking a lot of his time. He never scrimped on me with his time and effort; he never said ‘you are taking from my time’ and he never complained about my bothering him with my requests and meeting. He was always there whenever I needed him, never absent and always ready with a bright manager mind to give care and advice. Words are not enough to say how much I owe Phil of thanks, not even in my own language, Arabic. The only thing I can say is “Thank you Philip Gray very much”. If it was my choice, I would wish I could have some of the research and management skills that Philip Gray has.

Secondly, I would like to thank my second supervisor Ray Welland. I get surprised every time a PhD student talks to me about his second supervisor. One told me that he met him only once every three months, another once every year, and one told me he met his second supervisor after he had submitted his thesis to give him a copy!!! I was getting very surprised because Ray Welland gave me a totally different picture of the second supervisor. He gave me from his time that no supervisor gave his student. He was there at every meeting to give from his endless experience. He was the one who guided me with his experience to overcome the difficulties of this research. His ideas and advice were the candles that lit the dark gaps in the research paths. Ray Welland, I cannot provide you more than “Thank you very much”. If it was my choice, I would wish I could have some of the experience Ray Welland has. Without Philip Gray and Ray Welland, this work would never have appeared and I would never have had the skills I have now. They left their finger prints in my life. They taught me how to think, how to spot a problem, how to solve it and how to document a research.

I would like to thank my wife who stood beside me all over of this trip. I came to the University every day even on weekends. She never complained or

bothered. She was always beside me supporting and pushing towards my success. All over the four years we spent in Glasgow, her patience gave me the power and the motivation to finish this research. My wife, the only thing I can say to you is “Thank you very much”. If it was my choice, I would wish I could have some of the patience she has. Special thanks to my little angel, Layan. Although she is too young to support and always trying to prevent me from going to the University, she was pushing and supporting me to finish the work. Maybe because she wanted to go out and there were no time to do so during my PhD. In all cases, her existence gave my life a flavour and encouraged me to finish my thesis so I could spend more time with her and my wife.

I would like also to thank the greatest and the wisest man in this world, my father. He encouraged me and gave me advice every time we made a chat. His words and wisdom made me stronger and more confident that I could complete my PhD. His financial support kept me alive when the scholarship was not enough in some cases. For all of this I say “Thank you my father”. Of course this work would never even have started without the blessings and prayers of my mother, the greatest woman on earth. Although her worries always made me feel I was still too young to depend on myself, those worries encouraged me to solve my problems and finish the work so she could get some rest of her worries. Her words, encouragement, patience, support and prayers were one of the key factors of this thesis success. All what I can say to my mother is “Thank you very much greatest mother”. If it was my choice, I would have asked for some of my father’s wisdom.

Great thanks are forwarded to my father and mother in law. They pushed towards finishing my work with their prayers. I would like also to thank my brothers, sisters and grandmother. All of them were very supportive. I missed many family members and occasions with them during this PhD. I missed Tutu’s party. She proved that she is the cleverest in the family with an average of 95.4% at high school. I missed playing with Yazan and Abood my nephews and Zina my niece. I missed meeting my sister “Basma” who called and supported me from abroad with her husband Islam. I really appreciate that, so thank you very much. I missed the graduation party of my brother Dr. Osama from the Faculty of Medicine in Jordan.

His chat was very supportive for me. His high ethics and good-conduct can be an example to ambitious people.

My special gratitude goes to my University in Jordan, Applied Science Private University, its management, and all staff for their financial and spiritual support over all my period of study. My sincere thanks go to my department at Glasgow University represented by its heads of department Prof. Ray Welland and Prof. Joe Sventec for their financial support and for all the facilities they provided to help me in finishing this work.

I owe many thanks to Beverly. She taught anybody who met her a lesson in patience and bravery. Whenever I remember her, I see my hard work and patience are very little compared to hers. This gave me the support to work harder to finish this thesis.

During this PhD, I made new friends. Some of them really influenced my life with their encouragement and advice. I can here mention Youssef Gdura, Salem Jebriel, Ibrahim Maddy, Dr. Ibrahim Areef, Mahmoud Sofi and Fathy. These friends taught me to be patient and hard worker and how to believe and depend on God. A special Jordanian friend who was the closest in UK, although I have not met him on this land at all is Bilal Suwan. He was always the one that when I talked to, I felt comfortable. He was always supportive and the only one with whom I shared my research problems. We always were trying to share the problems so we could help each other to find solutions. In the near future we will be work colleagues. Another very special friend who shared me my work and thoughts for the past two years was Athanasios Pavlopoulos. He was a good friend and I spent with him very nice times that I would really miss. His patience was supportive of me and let me work harder. Thank you very much Athanasios for your time and support.

A very special person who supported me spiritually and financially was Sylvia Morgan. Sylvia is the nicest person I have ever met in UK. Her confidence, endless help, cheerful spirit and gorgeous smile were priceless supportive tools to finish my research. Thank you very much Sylvia. Another special person in my PhD life was Guido Zuccon. This very nice Italian guy taught me how to work hard and how to

remove the word “tired” from my dictionary. With his success, he was pushing me to finish my work.

I would also like to thank all the participants in my studies. Without them, I would not have finished this research. In particular, I would like to thank the friends who participated in the pilot studies, Athanasios Pavlopoulos, Mark Shannon, and Paul Keir. Their feedback was very helpful.

Special thanks deserve to go to Helen McNee. I still remember how I bothered her with my problems even before I started my PhD when I sent her an email asking for quick reply to my application. She was the first person who met me at Glasgow University on an unforgettable day. She never let me down although I asked her so many things. Special thanks also to Gail Rate. She is the one that manages all the work and I believe I was bothering her a lot in managing my work. Special thanks also to Elizabeth McFarlane. Although she was always busy, she always showed me her very nice and encouraging smile and she was always ready to help. I will not forget of course the technical support people whose help to fulfil my task continued until the last moment. All of these faculty staff made my study life in Glasgow easier, which reflected on my ability to finish this work. To all of them I say thank you.

I would also like to express my thanks to my friends in UK, Dr. Mohanad Alajlany, Omar Radwan, Mohammad Hassoneh, Firas Omar, Radwan Abu Jassar, Khaled Aljayyousi, Hani Aljanadily, Dr. Yousif Alshekh, and Dr. Yousif Qasrawi. Thanks are extended to my friends abroad Dr. Hisham Abusaimeh, Dr. Tareq Alhmidat, Mohammad Najar, Abdulrahman Abuloha, Mohammad Momani, Baha Sabah, Abdulrahman Mansi, Iyad Alsalamini, Hani Salameh, Sami Salameh, Mohammad Qattous, Dr. Mohammad Alazeh and to all other friends who supported me during this work.

Chapter 1

Introduction

1.1 Introduction

This chapter introduces the research presented in this dissertation which aims to simplify and facilitate constraint specification in meta-CASE tools for the purpose of specifying a CASE tool with its associated modelling language. The constraints introduced in this thesis consist of software engineering modelling language constraints such as those related to model element connections and cardinality. Specific examples of such constraints could be “It is not allowed to connect an Actor vertex with another Actor vertex using an Association edge type” and “It is not allowed to have more than one Start State vertex in a State Transition Diagram”. Such constraints specify the syntax and semantics of the software engineering modelling language.

The chapter details the different dimensions of the problem including the importance of CASE tools, the role of meta-CASE tools and the problem of constraint specification in such tools. It also proposes a solution based on a novel technique called *Constraint Specification by Example*. In addition to setting the problem context, this chapter presents:

- the aims and objectives of the research,
- an outline of the approach followed to achieve them,
- the contributions and achievements of this research in the problem context, and
- a summary of the dissertation, including a figure and a table presenting the dissertation structure, contributions and achievements.

1.2 Research Problem and its Context

In the field of software engineering, Computer-Aided Software Engineering (CASE) tools for diagram-based modelling play a role in facilitating the work of software engineers. CASE tools are helpful to software engineers in different ways at different software development stages (Henkel & Stirna, 2010). By contrast, several authors such as Iivari, (1996) and Kelter, Monecke, & Schild, (2009) have reported a problem of CASE tool inflexibility due to their generality. That is, generic tools don't

provide modelling elements or structures tailored to the particular needs of particular software development domains or contexts of use.

Consider the case of Domain Specific Languages (DSLs) that offer the ability to capture and model domain specific concepts that general languages cannot easily model (i.e., cannot capture the domain specific concepts) (Zschaler, Kolovos, Drivalos, Paige, & Rashid, 2010). An example of such DSLs is the modelling language required to model an instance of a network that follows the ZigBee network protocol. ZigBee defines the specification of the network layer that provides a framework to build applications in the application layer. ZigBee specifies rules that control the relations between different components of the network which also affects the topology of the network. To build a network depending on the ZigBee protocol, there is a need to follow the rules that it specifies. Because of the limited availability of specialised modelling tools that capture the semantics of ZigBee-based network configurations, researchers at a UK university developing ZigBee-based sensor networks use general drawing tools and even “pin & paper” physical graphs to model their systems. Clearly, a tailored modelling tool that captures the particular features and constraints of such networks would be desirable; in fact errors in the network specification have occurred because of the inadequacy of the modelling techniques. However, it is too costly for them to build an appropriate modelling tool or to have one built on their behalf¹.

In addition to the requirement of DSLs, in some cases, there is a need for local customisation of an existing tool. This means that the required tool for modelling the required diagram exists but there is still a need for some customisation of diagrams to suit local conventions, for example. One example of such modification in a diagram modelling tool could be the requirement of a Class Diagram editor to enforce starting the names of classes in the diagram with capital letters and starting the names of

¹ Personal communication with Loughborough University research fellow. September, 2010.

variables with small letters. C or C++ developers may require a tool that enforces an underscore prefixed to the names of class scope variables.

More generally, although often potentially useful to the software developer, domain specific tools are not widely or commercially available because of their poor cost-benefit profile (Ledeczi et al., 2001). That is, building domain specific tools is typically not cost effective given their limited context of use. Meta-CASE tools have emerged as a potential solution to this problem. Meta-CASE tools are tools that generate other tools (Gong, Scott, & Offen, 1997). They can be used to increase the flexibility of modelling and to generate the required domain specific tools and their associated DSLs (or formalisms) with lower cost and effort (De Lara & Vangheluwe, 2002). Such meta-CASE tools specify and generate CASE tools via a meta-modelling process which uses a meta-modelling language and generates, simply, a meta-model with its associated constraints.

In general, a modelling language can be specified by defining its vocabulary, syntax and semantics (Sommerville, Welland, & Beer, 1987) which is specified using a meta-modelling process. Meta-modelling process is the process that generates the meta-model. The meta-model is a composite of a model (could be a graphical meta-model as in the case of MetaEdit+) and constraints. The constraints are considered as an “important part of the metamodel” (Tolvanen, Pohjonen, & Kelly, 2007). However, some literature describes the constraints as additional information applied to the meta-model which is considered to be the diagram itself. This is the case, for example, when Ledeczi et al. (2001) differentiate between the diagram itself and the constraints by stating that the meta-model describes the DSL and specifies its syntax but not its semantics which can only be specified using the constraints (Ledeczi et al., 2001). De Lara & Vangheluwe (2002) clarify that the constraints are important extensions for specifying the modelling formalism and the purpose of using the constraints is to limit the number of meaningful models. Therefore, constraints have a vital role in modelling language specification through the meta-modelling process in meta-CASE tools.

Constraint definition is a difficult, time-consuming and error prone task that needs experience in the domain to be specified and expertise in the constraint language or technique associated with the meta-CASE tool being used (Ali, Hosking,

Huh, & Grundy, 2009; Groher, Reder, & Egyed, 2010). In addition to the previous documentation of this problem, the following argument shows that some empirical studies have been conducted to provide evidence of the difficulty of the constraint definition task.

Ackermann (2005) states that using Object Constraint Language (OCL) for specifying behaviour of software components is a “time consuming and error-prone” task. He refers to two case studies of using OCL in specifying business components. The results of these studies confirm that, although using OCL is useful for specification, “editing OCL constraints manually is nevertheless time consuming and error-prone”. Costal, Gomez, Queralt, Raventos, & Teniente (2006) detail the same problem of editing formal constraint languages in general and they add in addition to the time consuming and error-prone problems that formal constraints are difficult to understand by non-technical readers and difficult to be treated automatically in CASE tools. Fish, Hamie, & Howse (2010) introduce the same constraint specification problem of being time consuming and error-prone. They justify the existence of this problem as “typical specifications may contain numerous constraints, which in addition often state complex facts about the elements of the component’s model”. Briand, Labiche, Di Penta, & Yan-Bondoc (2005) conducted a controlled experiment to evaluate the usefulness of OCL combined with UML diagrams. The results indicated the difficulty of using OCL limits its usefulness based on the “ability, experience, and training of software engineers” in using it. Barr (2000) reported an empirical study which uncovered the difficulty of using OCL and constraint formal languages in general. The difficulty is represented in the mistakes that the users made throughout the experimental task of specifying constraints using OCL. Some of the problems that were documented and participated in the difficulty of OCL are “excessive complexity” and redundancy of OCL, “misunderstanding of OCL semantics”, “unclear issues in OCL semantics” and “insufficient semantical interconnections”.

In the context of meta-CASE tools, Liu, Hosking, & Grundy (2007a) claim that “most want to avoid having to use textual scripting languages or programming language approach directly” for diagram editors behaviour specification including constraint specification. This is an indication of such specification complexity. In

another paper, Liu, Hosking, & Grundy (2007b) document that “an area that commonly proves difficult for meta-tool designers is the specification of model level behaviours, such as semantics, constraints, dependencies, element initialisations, calculations, etc. Most approaches for model behaviour specifications use conventional code in the form of event handlers or constraint expressions”. Bergmann, et al. (2010) claim that the Eclipse Modelling Framework (EMF) utilises model queries including constraints for domain specific languages. They include using OCL as an example of the model queries. However, they also include a limitation of such model queries which is their complexity and time consuming nature. They claim that this limitation is based on the industrial experience of the authors.

In the context of drawing editors Alpert (1993) observes that layout constraints are useful in such editors but pointed out its complexity by stating that “the challenge remains of how to facilitate constraint specification”. He introduced programming by demonstration as a solution for the constraint specification complexity problem and claimed that this technique is considered as “simple and natural”.

Bimbo & Vicario (1995) see the problem in transforming the designer’s empirical understanding and experience of the domain specification to a textual abstract representation form. The problem of the complexity of constraint definition has already been addressed by others. Some approaches include using a general purpose programming language (e.g., Java) instead of a constraint programming language (Zhu, Grundy, & Hosking, 2004), using a visual programming language (Liu, Hosking, & Grundy, 2007a), or employing a spreadsheet-like interface (Li, Hosking, & Grundy, 2009) and the form-filling technique, found in some meta-CASE tools such as MetaBuilder (Gong, Scott, & Offen, 1997) and the commercial meta-CASE tool MetaEdit+ (MetaCase, 2009).

Nevertheless, constraint definition remains a research challenge. The solutions described above require that a tool developer, typically a software engineer with knowledge of the modelling domain but little or no experience with CASE tool building, must become an expert in a complex constraint language or constraint specification tool. Ideally, such a developer of a domain specific tool should be able to produce their tool without investing considerable effort in learning how to use the

meta-CASE system. The research reported here offers an advance towards that ideal by introducing a novel technique that can reduce the difficulty of the constraint specification task.

Research Problem:

Constraint Specification is a difficult task because it is error-prone, time consuming and in meta-CASE tools there is a gap between the specification domain (as text) and the application domain (as modelling language).

Research Question:

Is it possible to reduce the difficulty of constraint specification in meta-CASE tools for the purpose of diagram editor specification?

1.3 The Proposed Solution

This research proposes a new technique, called *Constraint Specification by Example* (hereafter, *CSBE*), based on the Programming by Example (PBE) technique as a solution for the problem of constraint definition complexity in meta-CASE tools. PBE or, Programming by Demonstration (PBD), depends on introducing examples of data and values to a system that generalises the example(s) and generates a program (Myers, 1993). This technique was originally invented to make programming an easier task and more accessible to non-programmers. PBE has been applied in different contexts such as document generation, robotics and, in contexts other than meta-CASE tools, for constraint specification. However, it has not been introduced or tested as a possible solution for the constraint specification complexity problem in the meta-CASE tools domain.

The CSBE technique has been developed to introduce an implementation perspective of PBE in the context of the constraint specification process in a meta-

CASE tool. The CSBE technique describes and implements a *synergistic*² relationship between the user and the tool to solve the constraint specification problem. This synergism depends in its simplest form on the user to introduce one or more examples that express the required constraint. From these examples, the system tries to infer the required constraint. The system depends on a rule-based inference engine for this purpose. Since any constraint can be expressed either using positive or negative examples (this is called example polarity), CSBE allows the user to introduce the constraint examples in different polarities, positive or negative. Finally, it gives the user the ability to customise and personalise the tool by augmenting the meta-CASE tool inference engine using a learning technique. It is believed that inventing CSBE will have a positive effect on reducing constraint specification complexity in the domain of meta-CASE tools.

1.4 Aims and Thesis Statement

The broad aim of this research is to simplify and facilitate the constraint definition task which is a part of the meta-modelling process for CASE tool specification in a meta-CASE tool. In particular, the work focuses on facilitating and simplifying the constraint specification task using a novel technique, called Constraint Specification by Example (CSBE), developed as part of this research and based on the Programming by Example (PBE) technique. This research focuses on a specific category of CASE tools, software modelling tools or diagram editors; for simplicity they are referred to throughout this research as CASE tools, however. This research, in general, sets out and argues for the following thesis statement:

² *syn·er·gy*/'sinərjē/

Noun: The interaction or cooperation of two or more organizations, substances, or other agents to produce a combined effect greater than the sum of their separate effects

Thesis Statement:

It is possible to simplify and facilitate the constraint specification task in a meta-CASE tool using the CSBE technique.

There are three research questions that follow from the thesis statement. These questions with the relevant research objectives are as follows:

Does CSBE improve the performance of constraint specification in a meta-CASE tool compared to the form-filling technique? One objective of this research is to study this criterion for the CSBE technique for constraint specification task in comparison with the typical form-filling constraint specification technique. This objective tests the claim that *the performance of specifying constraints in a meta-CASE tool using the CSBE technique is higher compared to the form-filling technique because the CSBE increases the effectiveness, efficiency and user satisfaction.*

Does example polarity influence the performance of CSBE? Answering this question is another objective of this research as it requires studying the effect of example expression polarity, positive and negative, on the performance of CSBE. This study validates the claim that *example expression polarity influences the CSBE technique performance as it improves the technique performance by allowing constraints to be expressed using the two available example polarities, positive and negative, instead of depending on one polarity.* This study is supported by another one studying the effect of the polarity of expressing the constraints using natural English language on understanding the constraints.

Does implementing and using the learning technique influence the performance of CSBE technique? The research, by answering this question, explores the feasibility and desirability of customising the CSBE inference engine using a learning technique. This study verifies the claim that *implementing a learning technique for augmenting and customising the knowledge base of the system improves the CSBE technique performance by increasing its effectiveness, efficiency, and user satisfaction, and thus facilitates the constraint specification task.*

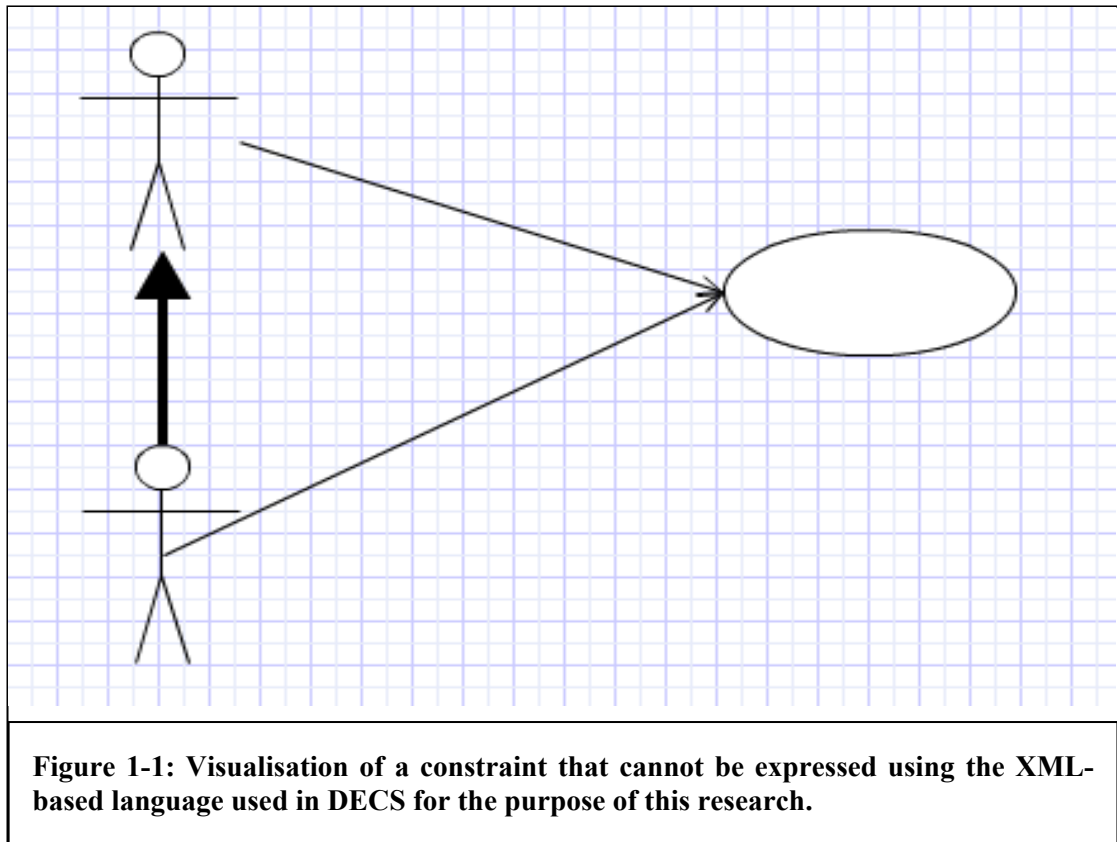
1.5 Approach

For the purposes of this research and, in particular, to validate the claims stated above, the Diagram Editor Constraints System (DECS), a meta-CASE tool developed at Glasgow University before the start of this research, has been used as a starting point. In addition, an informal XML-based constraint language with associated parser and constraint checker has been developed and implemented together as a separate component in DECS. The developed constraint language is able to specify several types of software design constraints including:

- Connectivity constraints that constraint connection between two specific vertex types. This includes the upper and lower bound numbers of edges outgoing or incoming from a specific vertex type.
- Vertices and edges labels-related constraints that are in the context of uniqueness and regular expressions.
- Cardinality constraints that limit the number of vertices, edges or structures composed of vertices and edges to specific upper bound number or lower bound number.
- Visual representation uniqueness for vertices and edges.
- Path-related constraints which include cyclic graph restriction and the existence of path between a specific vertex to some other vertices in the graph.
- The language also is able to specify constraints over vertex and edge properties such as the background colour, the font colour, and the decoration (the arrow head in case of edges).

The language was designed to be able to specify all the required constraint types used throughout this research since building and using a complete constraint language is out of scope of this research. However, the constraint language is not complete as it lacks the ability to specify all the constraints that formal constraint languages can specify such as the constraint “it is not allowed for a vertex of type Actor to be connected with a Use Case that is connected to another Actor that is connected to the first Actor using a Generalisation edge type”. This constraint

(appears in Figure 1-1) requires the concept of “self” that exist in OCL but not in the language used in this research.



However, because the constraint language developed for this research has some distinctive features that will be discussed in detail in Chapter 3, it has been considered as an achievement of this research. Developing and improving the capabilities of this immature constraint language has also been introduced as an area for future work. The current version of DECS depends mainly on constraints for CASE tool specification. It defines the target modelling language by specifying the constraints that control the behaviour of the designer in the generated editor (the target CASE tool).

To answer the research questions and achieve the objectives stated above, two constraint specification techniques have been developed in DECS, the form-filling technique, represented as a wizard and tabbed forms, and CSBE. The form-filling technique has been selected because it is a typical technique used for constraint definition in documented meta-CASE tools, apart from text-based approach; it is used in meta-CASE tools for the purpose of constraint specification and it is common in

other different contexts which supports its familiarity. To implement the CSBE technique a rule-based inference engine has been adapted from Sazonov (2004) with several modifications and implemented as a separate component, *the inference manager*.

To investigate the first research question, an empirical study has been conducted comparing the two implemented techniques, viz., form-filling and CSBE, with respect to effectiveness, efficiency and user preference in constraint specification. This study offers a contribution of this research as neither the PBE technique nor any of its derivatives, such as CSBE here, has been used before in the context of meta-CASE tools. It is also a contribution in the field of PBE itself as PBE has never been used before (according to the reviewed literature) for software engineering modelling-related constraint specification.

For the purpose of studying the second research question, a study has been conducted comparing two implementations of CSBE; the first allows the user to express the required constraint using either of the two available example polarities, positive or negative, while the second implementation allows the constraint to be expressed using only one example polarity (viz., negative). Although example polarity is an associated feature to almost all examples in the field of PBE, this feature has not been studied empirically before, which is considered a contribution in this research. Prior to this study, another supportive study has been conducted to explore user preference and the effect, in terms of comprehensibility, of expressing constraints in different forms in natural language.

Finally, an empirical study has also been conducted to evaluate the feasibility, desirability and added value, of enabling users to add and customise the inference rules used by the CSBE technique. This study is used to investigate the third research question stated above. Such a feature has not been introduced or implemented before in any PBE system, which can be considered a significant contribution in this field.

1.6 Chapter Summary

This chapter has introduced the problem that motivates this research. It describes the importance of CASE tools and the requirement of the ability to support

domain specificity. It justifies the cost inefficiency of developing such domain specific CASE tools. For this problem, meta-CASE tools have been introduced as a solution. However they suffer from the difficulty of the tool specification process, especially the constraint specification task. The chapter introduces a proposed solution for the constraint specification difficulty problem. The solution is built around a novel constraint specification technique, Constraint Specification by Example (CSBE), which is based on Programming by Example (PBE).

The chapter also introduces the aim of the research of simplifying and facilitating constraint specification in the domain of meta-CASE tools. To achieve this aim, objectives of the research set out the research claims and questions. The approaches to validate these claims are discussed in detail. These approaches can be summarised as three novel ideas associated with studies for evaluation. The first study implements the CSBE technique in a meta-CASE tool that generates constraint-based software engineering modelling editors. A study was conducted to evaluate the performance of this technique in terms of effectiveness, efficiency and user satisfaction compared to another implemented typical constraint specification technique, the form-filling. The other two studies, availability of example polarities and the tool customisation through a learning technique, were conducted to evaluate the influence of these two features and their implementations on the performance of CSBE in facilitating the constraint definition task.

1.7 Dissertation Roadmap

Chapter Two reviews the literature and provides the required background in the area of meta-CASE tools and PBE. The required definitions and importance of CASE tools, domain specific languages, and meta-CASE tools with its meta-modelling process will be introduced. The vital role of constraints in the meta-modelling process and in the generated modelling editors is detailed with a review of the most related constraint classifications. The review also includes the different techniques of constraint specification in the domain of meta-CASE tools. The chapter also reviews some PBE systems in different domains.

Chapter Three describes DECS in its original form, as it was at the start of this research, and the enhancements introduced to it; that is, the chapter presents DECS as a context for this research. The chapter details DECS' structure and focuses on the constraint manager component with some technical implementation details. It also introduces the XML-based constraint language with its implementation which facilitates building complicated constraints (that have many vertices and edges participating in them) in a flexible way. Finally, the chapter presents the form-filling technique and relates it to the constraint language which sets out the flexibility of this technique in constraint specification.

The fourth chapter details the theoretical synergistic model of the CSBE technique that has been proposed as a solution for the problem. The chapter continues in describing this model through presenting CSBE and its distinctive features. The chapter also describes the rule-based inference engine and the visual generalisation feature (also referred as the remodelling feature). These feature discussions are supported with screenshots for better understanding.

The following three chapters detail the three empirical studies with a fourth small related study associated with the second one. Each chapter presents its empirical study by giving its aim, hypothesis, design, procedure, results, discussion, threats to validity and finally, a comparison with related work. The eighth chapter summarises and concludes with a discussion of the relation between the conducted studies and the main aim and objectives of the research. Finally, the chapter proposes some ideas for future work.

Figure 1-2 presents a “roadmap” of the thesis followed by a clarification of the achievements and contributions in each chapter. Some chapters are divided into two parts to distinguish the contents in terms of the contributions and achievements.

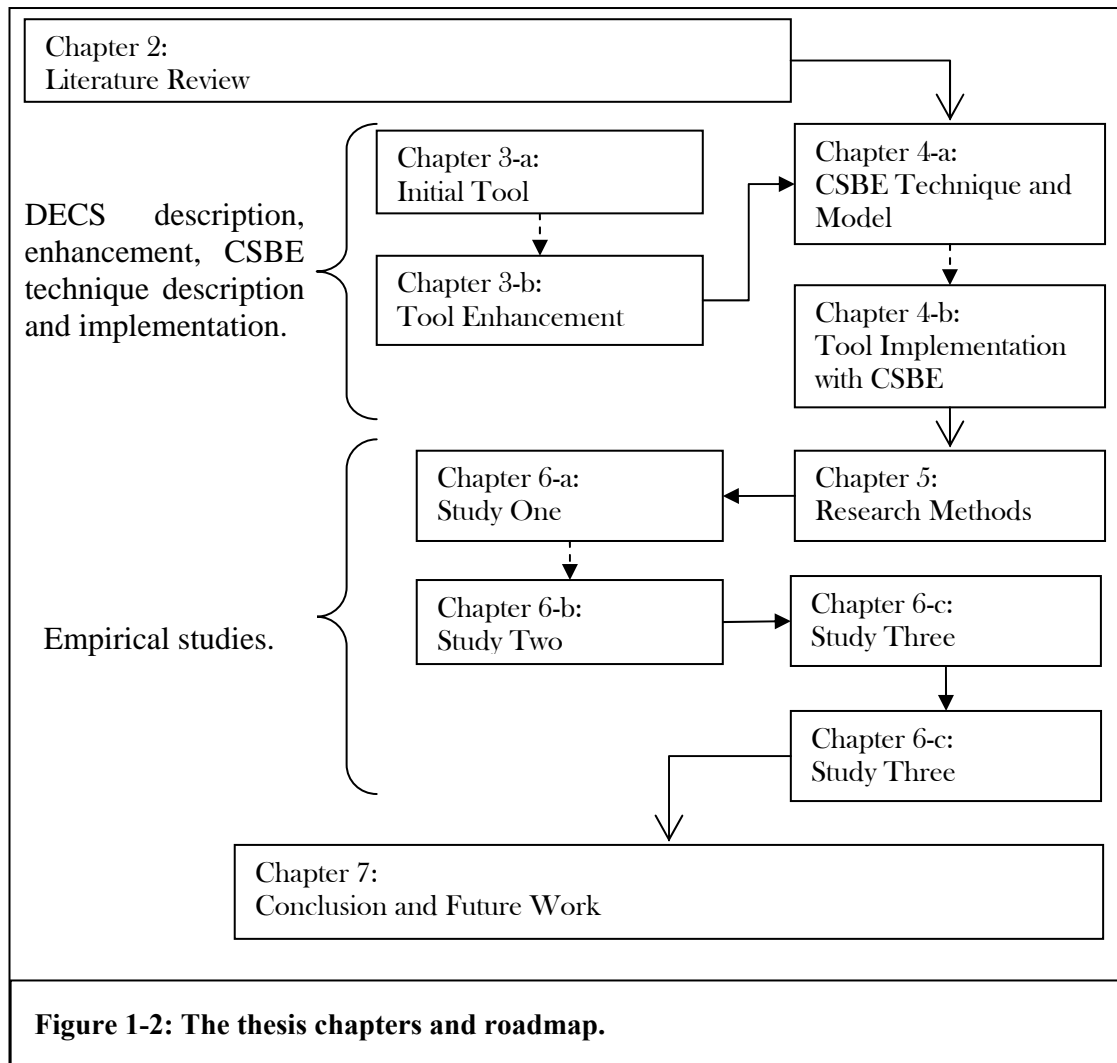


Table 1-1 shows a summary of the contributions while Table 1-2 shows a summary of the achievements found in the chapters presented in Figure 1-2.

Table 1-1: Contributions distributed over thesis chapters

Chapter	Contributions
Chapter 4-a	<ul style="list-style-type: none"> • A novel specification technique, CSBE, for constraint specification in meta-CASE tools.
Chapter 6-a	<ul style="list-style-type: none"> • Demonstration, via an empirical study, that CSBE is superior to a wizard-based form-filling technique.
Chapter 6-b	<ul style="list-style-type: none"> • An empirical study of the relative comprehensibility of constraints expressed negatively vs. those expressed positively in a natural language.
Chapter 6-c	<ul style="list-style-type: none"> • Demonstration, via an empirical study, that the use of a multi-polarity technique vs. a uni-polarity technique improves CSBE performance.
Chapter 6-d	<ul style="list-style-type: none"> • Development of a novel rule augmentation technique for CSBE. AND • Demonstration, via an empirical study, that adding a rule augmentation facility to a CSBE system improves performance.

Table 1-2: Achievements distributed over thesis chapters.

Chapter	Achievements
Chapter 2	<ul style="list-style-type: none">• A literature review of:<ul style="list-style-type: none">◦ meta-CASE tools, their use of constraints, the characterisation and classification of constraints and the methods by which they are defined,◦ PBE with its different application contexts, use of example polarities and techniques for rule learning.
Chapter 3-b	<ul style="list-style-type: none">• Enhancements to the DECS meta-CASE system which made it suitable to be used as a prototype for this research.• Design and development of an XML-based constraint language used in the studies in this research. This language has many features, such as flexibility, which qualified it to be adopted in this research.• Development and implementation in DECS of a constraint management component that handles constraints specified in the constraint language described above.
Chapters 4-6	<ul style="list-style-type: none">• Implementation of CSBE in DECS with all of its distinctive features.

Chapter 2

Background

2.1 Introduction

This chapter explores the different aspects of the background to this research and reviews the literature in the domains related to it. The chapter starts by giving an overview of Computer Aided Software Engineering (CASE) tools and their importance. Then it identifies some problems that limit the user of these tools, viz., the requirement for domain specificity and customisation, and provides a solution, meta-CASE tools. It describes meta-CASE tools and the way in which they enable CASE tools to be defined and generated. The chapter then explores the importance and role of constraints in CASE tools and in specifying CASE tools using meta-CASE tools. This section ends by reviewing the literature documenting the difficulty of constraint specification, which clarifies the research problem, and which examines the techniques used to define constraints in meta-CASE tools.

The chapter then reviews the domain of the proposed solution, Programming by Example (PBE), and provides examples of using this technique in different domains. The chapter also explores different aspects of PBE such as example polarity and addition of inference rules.

2.2 Computer Aided Software Engineering (CASE) Tools

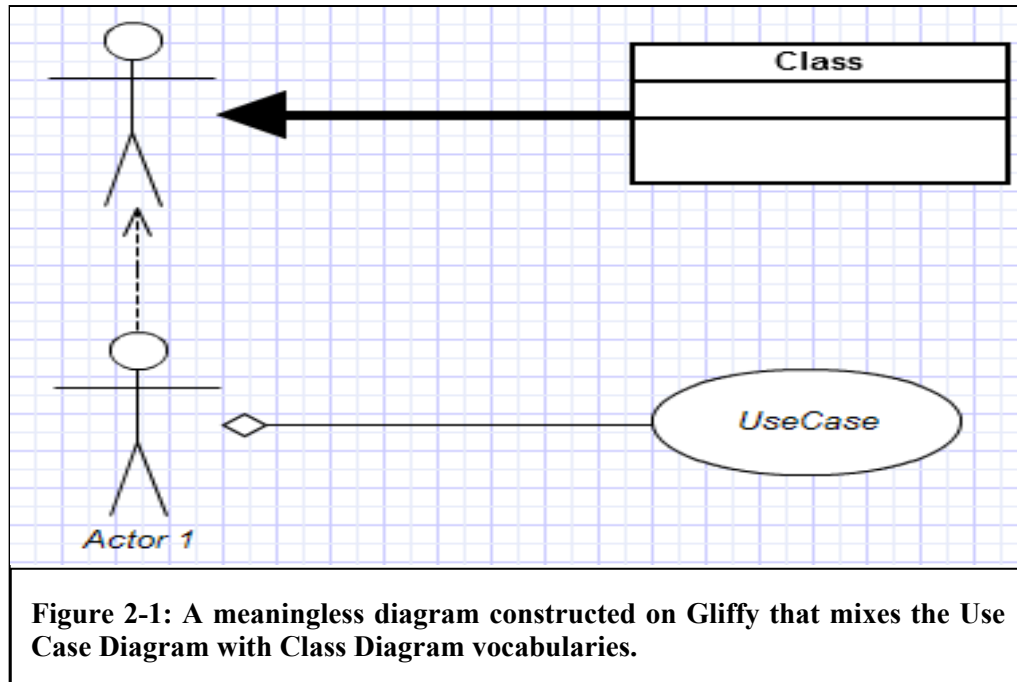
CASE tools are programs that support software engineering process activities and provide a wide range of services such as requirements analysis, design, model editing, documentation and report generation, code generation, and testing (Sommerville, 2007 page 12). CASE tools are helpful to software engineers in increasing productivity, improving control of the development process, shortening development time, and improving software quality. This reduces software production and maintenance costs and increases customer satisfaction (Henkel & Stirna, 2010). Diagram editors, as a type of CASE tools, are usually associated with, and used as a support for, software engineering methods for the purpose of developing graphical models that describe the system (Sommerville, 2007, page 12). “Method” is the term used to describe one or more activities of a software development life cycle (Alderson, 1991). These methods are techniques for describing “software specifications” in different information representations starting from source text to

graphical representation (Findeisen, 1994; Ledeczi, Maroti, & Volgyesi, 2001). Sommerville (2007 page 12) claims that each method should have four components. These include descriptions of the system models and their notations, constraints that specify the system models, recommendations that lead to good design, and description for the order of activities in the method. CASE tools support methods by providing model editors that use the method's notation (Sommerville, 2007 page 12; Gong, Scott, & Offen, 1997).

'CASE tool' is a term that covers a wide range of different types of tools. Sommerville (2007 page 87) introduced a classification of CASE tools depending on their functions. One category of this classification is editing tools such as text editors, diagram editors, and word processors. This research focuses on a specific category of CASE tools, viz., software engineering graph-based diagram (or graph-based model) editing tools, which lies within the editors class of CASE tools according to Sommerville's classification. For simplicity, software engineering diagram editing CASE tools will be referred to as 'diagram editors' throughout this document.

However, software modelling editors come in a variety of forms, with different levels of functionality. Software diagram-based modelling tools can offer a range of functions to a developer starting from general graphical output in tools like Gliffy (Gliffy, 2011) through a suite of specialised editors for different modelling techniques as in a system like Rational Rose ® (IBM, 2009). The last is a tool that provides the user with different Unified Modelling Language (UML) diagram editors for different purposes such as a Class Diagram editor and a Sequence Diagram Editor.

In tools like Gliffy, the diagramming vocabulary is typically supported along with the functions for joining objects via edges into graph-like structures with no constraints governing rules of the method, which was introduced as one of the components of any method by (Sommerville, 2007 page 12). Some examples of the required constraints are "connection constraints" (it is not allowed to connect two vertices of specific types), "cardinality constraints" (not allowed to have more than 3 vertices of specific type in the diagram) or "representation constraints" (labels of vertices and edges must be unique in the diagram).



This is shown in Figure 2-1, which shows a diagram composed of several different methods' notations related to two different diagram types, Class diagram and Use Case diagram, constructed using Gliffy. Other tools such as Rational Rose allow the enforcement of constraints on legal diagrams arising from the semantics of the software model being represented. Such tools do not allow the meaningless ad hoc diagram that appears in Figure 2-1. Based on this difference, diagram editors can be categorised into two types, those that enforces constraints and those that do not. This research focuses on the specific category of software graph-based modelling editors which enforce semantic constraints on the graphical models (or diagrams) that are allowed to be produced.

Examples of the type of diagram editors that this research is focusing on are State Transition Diagram editors and Use Case Diagram editors. Minas (2002) supports categorising diagram editors and defines diagram editors as graphical editors that are designed or tailored for a specific diagram language. He distinguished the category of diagram editors from that of general drawing tools in terms of the ability of diagram editors to understand the distinctive features of the diagrams produced using them; it follows that they do not allow a user to produce arbitrary drawings, as in Figure 2-1. Instead, they are restricted to visual components allowed in the diagram language. He also introduced a Class Diagram editor as an example of the

class of diagram editors and stated that it is not allowed to use it to draw a transistor symbol which is possible in a diagram editor for circuit modelling.

2.3 CASE Tools Limitations

As introduced above, CASE tools and in particular diagram editors are helpful to software developers and currently many of them are available commercially in the market. Many diagram editors are very useful especially as editors for UML diagrams. Although of benefit to software engineers, many diagram editors have problems that reduce and limit their use. Reviewing the literature revealed some problems that can be summarised as follows:

- **Customisation Difficulty**

Typically, no customisation for the behaviour or the modelled language is available. Accordingly, the users must accept and learn what the tool provides. They should not have high expectations of modifying the tool to suit their preferences. In other words, the user of these tools such as Rational Rose, should accept to generate UML diagrams with formal UML and follow the rules set by the tool developer. Customisation of Rational Rose to be able to draw use cases with different vertices to give different priorities for them in the project, as an example, is not possible. Again, such tools are helpful as a general diagram editors but their user should accept what it offers.

- **Fixed Methodology**

Most diagram editors support fixed methodologies with a fixed set of methods; they provide only limited support for domain specific concepts. This is highly related to lack of customisation in currently available diagram editors. This summarises the problem that diagram editors are programs that are hard coded by vendors. For commercial purposes, each vendor developed one or more diagram editors trying to make them as general as possible to be used in different domains. Moreover, vendors developed diagram editors to support used methods instead of a user's domain specific methods. This creates diagram editors that cannot be modified or customised to suit a specific domain instead of being general tool or not capable of supporting a domain that the user is interested in modelling. If this problem is applied to Rational

Rose again, the tool provides only the ability to draw UML diagrams. This means that it sticks to an approach that its vendor supports. This approach and its methods are general enough as the software engineers usually require the UML in projects that depend on object oriented architecture. However, the user will not be able to invent a new diagram into it or remove a diagram from its installation. As an example, the user will not be able to use Rational Rose to create a diagram for a network in a building. The user must accept it with its supported methods.

- **Users Must Adapt to Tool**

Diagram editors force their users to be customised to them not vice versa. This problem emerged as a result of the above two problems. Users have to work with, and adhere to, what is provided by the available tools with their supported methods. This, obviously, limits the options for modelling a domain or a software system to those made available via the paradigm of the tool being used (Marttiin, Rossi, Tahvanainen, & Lyytinen, 1993).

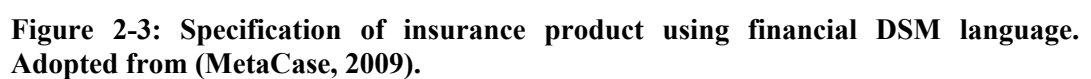
These problems reduce, to an extent, CASE tool use and increase the demand for domain specific and customisable modelling tools. One of the main advantages of diagram editors is their potential domain specificity; however, this feature does not always exist in currently available CASE tools (Kelter, Monecke, & Schild, 2009).

2.4 Domain Specific Languages and Tools

The solution for the problems identified above is providing domain specificity and customisability in CASE tools through support for domain specific modelling languages and CASE tool customisation. A Domain Specific Modelling Language (DSML), for simplicity, Domain Specific Language (DSL), is a modelling language that can capture the specific domain concepts that it was designed for, the thing that general modelling languages lack. A DSL captures and models the domain specific concepts using its domain specific vocabulary, syntax and semantics. By being able to construct domain specific models, a DSL is an effective way of facilitating the task of application development and improving the productivity of software engineers (Goldman & Balzer, 1999; Santos, Koskimies, & Lopes, 2010; and Kelly & Tolvanen, 2008). Zschaler, Kolovos, Drivalos, Paige, & Rashid (2010) and Kirchner & Jung

(2007) document the same point by emphasising that the purpose of constructing and using a DSL is to tailor the modelling language capabilities for the specific domain required to be modelled.

Tools that support DSL are called domain specific tools. They are diagram editors that allow the user to build models using the DSL. Accordingly, domain specific tools are considered as a subclass of CASE tools. Domain specific tools capture the specifications of a domain in the form of domain specific models, enhance specific activities of the software development process such as accelerating the activity of requirements engineering and bridge the gap between the application and implementation domains (Ledeczi, Maroti, & Volgyesi, 2001; Atkinson, Gutheil, & Kennel, 2009; Guo, Sierszecki, & Angelov, 2009; Robert & Bernhard, 2005). In an attempt to evaluate the need and the availability of domain specific modelling tools for commercial use, the MetaCASE website (MetaCase, 2009) was investigated. It was found that commercial DSLs and their associated domain specific modelling tools are used in a number of different domains. According to the website, the purpose of such domain specific tools is to improve the productivity and the quality of the industrial software development projects. In total 20 domain specific modelling tools are presented on the website as examples of the tools that have been built and that are being used. Examples of the domains of such modelling tools include mobile applications, car infotainment systems, a DSL for railway track control and the design of web applications. A comment from “Burton Group” on the same website says “The use of domain-specific languages and custom meta models is the greatest aid to productivity and making model-driven development a viable practice. Unfortunately, most vendors ship general-purpose UML modelling tools that are not easily customized to reflect domain-specific notations and constructs”. This provides some evidence of the importance and wide scale of use of commercial domain specific modelling tools. The following figures (Figure 2-2, Figure 2-3, Figure 2-4, Figure 2-5) show some examples of domain specific languages and their associated modelling tools developed using MetaEdit+. All the figures are adopted from (MetaCase, 2009).



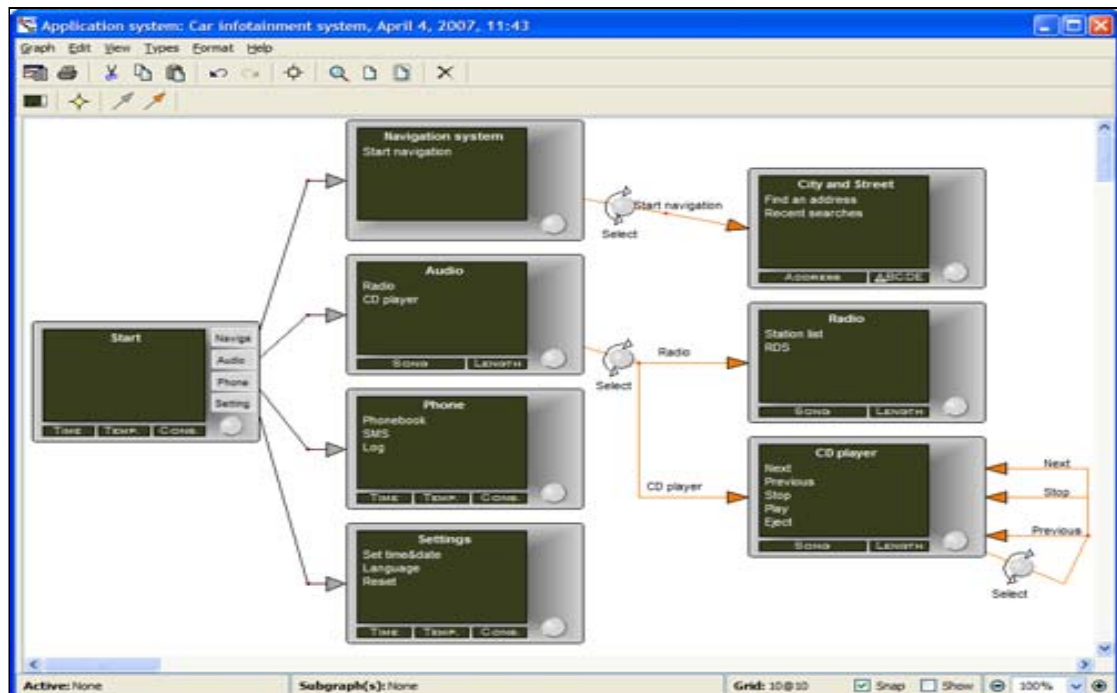


Figure 2-4: DSM language for designing car infotainment systems. Adopted from (MetaCase, 2009).

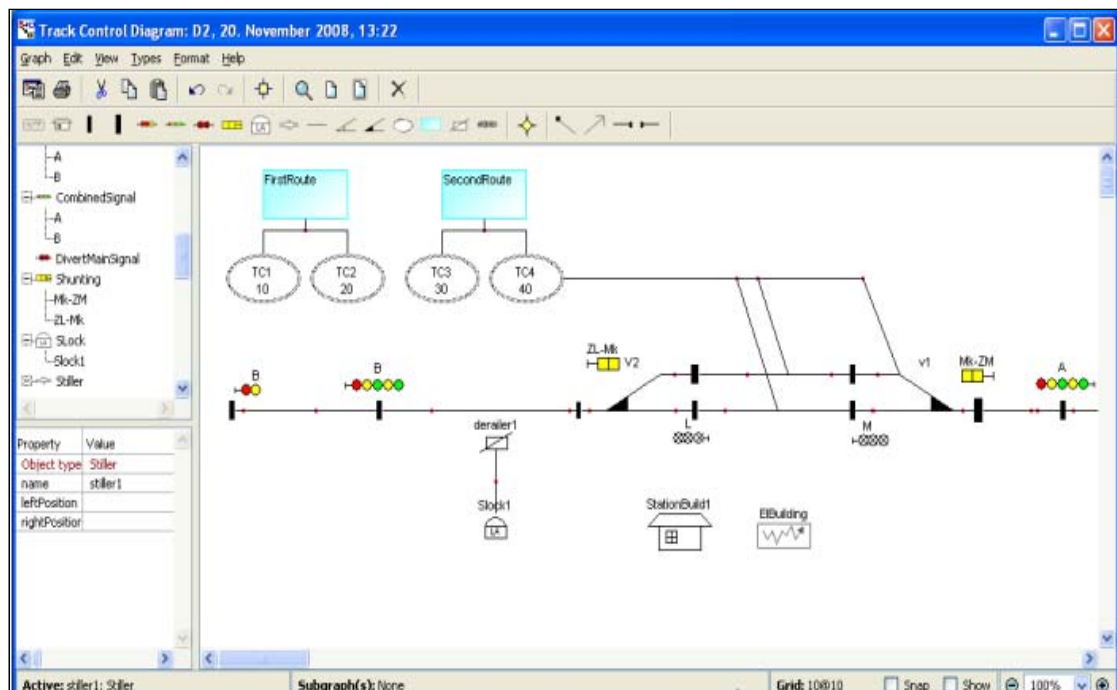


Figure 2-5: DSM language for modelling the layout of the railway track. Adopted from (MetaCase, 2009).

Although they are useful, domain specific diagram editors suffer from a problem that limits their use. They are considered expensive, with a poor cost-benefit ratio because each domain requires a specific tool to be built starting afresh specifically for it. This requires considerable effort over a significant period of time (Goldman & Balzer, 1999; Ledeczi et al., 2001; Ledeczi, Maroti, & Volgyesi, 2001; and Kelter, Monecke, & Schild, 2009). Engstrom & Krueger (2000) also introduce another problem which is the quick development and changes in the domain which requires a continuous development of the domain specific tool. Consequently, domain specific tools are available only for widely used domains with large markets that justify the investment. This justifies the problem of unavailability of domain specific diagram editors commercially (Ledeczi, Maroti, & Volgyesi, 2001; and Gong, Scott, & Offen, 1997). Meta-CASE tools, or CASE shells as Marttiin, Rossi, Tahvanainen, & Lyytinen, (1993) call them, solved this problem by their ability to specify and generate domain specific diagram editors including software engineering design diagram editors, the domain of this research.

2.5 Meta-CASE Tools

Ledeczi, Maroti, & Volgyesi (2001) introduced the Generic Modelling Environment (GME) as a meta-CASE tool that generates “graphical modelling” editors. They introduced the meta-CASE tool as a solution for the problem of DSL diagram editors and the problem of creating customised diagram editors. The meta-CASE tool specifies and generates diagram editors that are tailored to the concepts of domains and customised as required. In other words, they generate what has been introduced above as domain specific diagram editors and also generate customised diagram editors as the user requires. It is claimed that these tools are generated and customised with less effort, cost and human resources than that required to build them from the scratch (Kelter, Monecke, & Schild, 2009; Alderson, 1991; Gray & Welland, 1999; and De Lara & Vangheluwe, 2002). Examples of the domain specific diagram editors generated using the meta-CASE tool GME are shown in Figure 2-6-a and Figure 2-6-b. Figure 2-6-a presents a domain specific diagram editor for modelling signal flow on a chip (Systems, 2011) while Figure 2-6-b shows a domain specific diagram editor for modelling a finite state machine (Ledeczi, Maroti, & Volgyesi, 2004).

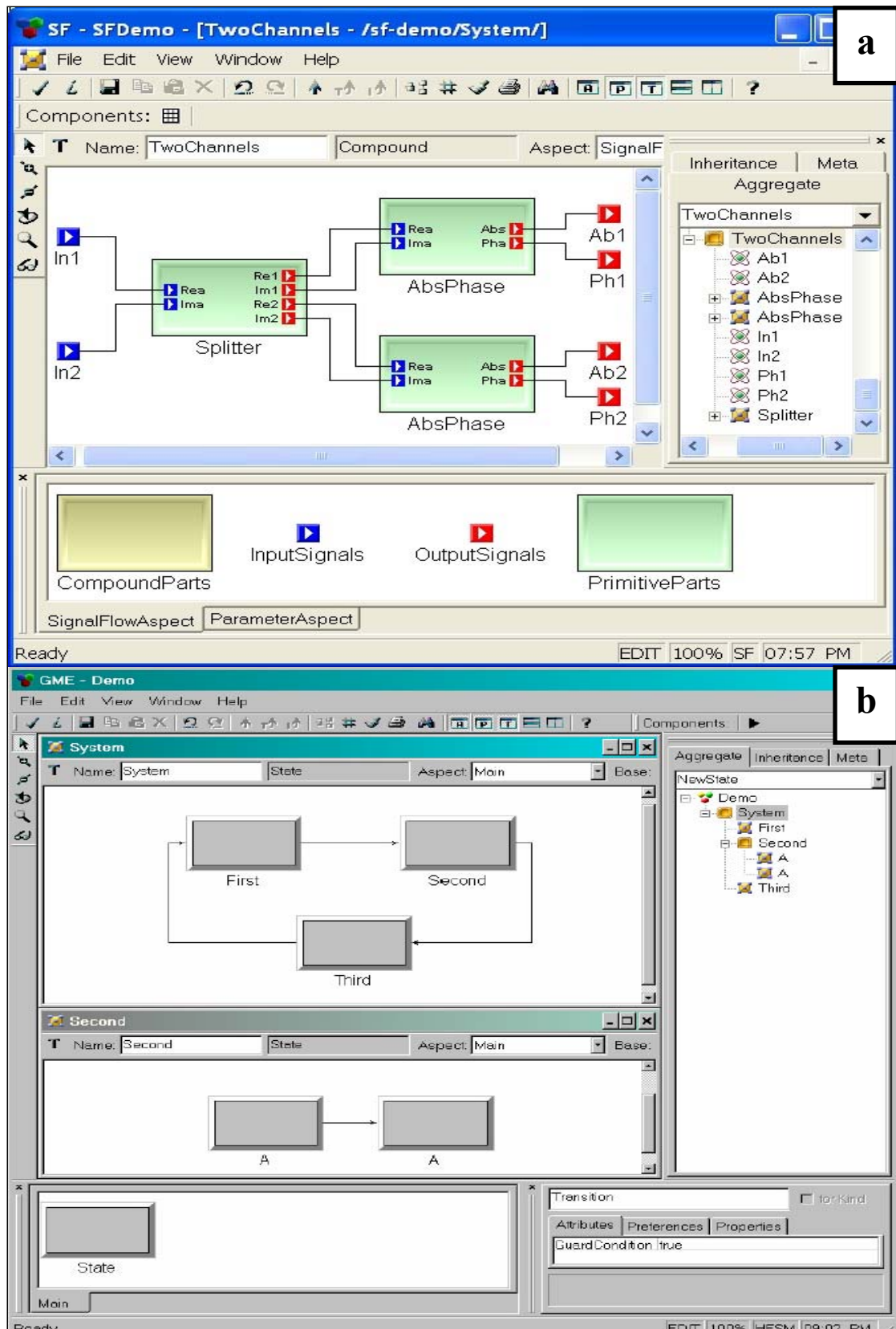


Figure 2-6: a) A domain specific diagram editor for modelling signal flow on a chip design environment generated using the meta-CASE tool GME. Adopted from (Systems, 2011). b) A domain specific diagram editor for modelling finite state machine. Adopted from (Ledeczi, Maroti, & Volgyesi, 2004).

Isazadeh & Lamb (1997) and De Lara & Vangheluwe, (2002) define meta-CASE tools as tools that can define, construct, and generate customised CASE tools to support specific development method(s). As a consequence, any meta-CASE tool is open for defining a number of different software development methods (Findeisen, 1994). The main reason behind using meta-CASE tools is their ability to provide cost-effective way to develop methods through generating its associated graphical modelling diagram editors. A meta-CASE tool is used to specify a CASE tool required to help in constructing graph-based models for a specific method or a specific domain. The user of the meta-CASE tool achieves this by specifying the modelling language itself (called the DSL or the target language) that the CASE tool will use to construct the models. In general, the user of the meta-CASE tool specifies the vertices (nodes) and the edges (connectors between the vertices) with specific visual representations that participate, and will exist, in the target language. Then the user specifies the constraints that should exist in the target language. Finally, the meta-CASE tool can generate the required tool as specified to be used by the designer to construct the required models using the target DSL. The process of specifying the target language is called meta-modelling and will be discussed in detail in Section 2.6 below. Comparing the time and cost of producing a domain specific CASE tool with and without using a meta-CASE tool shows the importance of meta-CASE tools. They are considerably faster and cheaper.

Ledeczi, Maroti, & Volgyesi (2001) claim that building a CASE tool using their meta-CASE tool (GME) takes from hours to one day which also encourages evolving new methodologies. Their meta-CASE tool, GME, uses a meta-model to specify the domain specific modelling paradigm (domain syntactic, semantic and presentation information) that defines the set of models that the target modelling environment can construct. The modelling paradigm information is used to construct the concepts of the domain, relationships allowed between these concepts, the presentation of these concepts and the rules (constraints) controlling the legally available model constructs in the target modelling environment. Atkinson, Gutheil, & Kennel (2009) identify the meta-CASE tool's role as allowing the users to generate (DSL) engineering tools.

The above discussion shows that meta-CASE tools are considered as a solution for the problems of DSL unavailability and the effort required to build tools from scratch. Meta-CASE tools solve the problems mainly by the ability of one meta-CASE tool to specify and generate several DSLs with their associated modelling editors. It is believed that this reduces the cost and effort required to build each domain specific modelling tool from scratch. Ledeczki, Maroti, & Volgyesi (2001) claim that the use of a meta-CASE tool reduced the time required to specify and generate a domain modelling tool to one day. One comment from the “Butler Group” on the MetaEdit+ website states: “MetaCASE, through MetaEdit+, provides a DSM tool for full code generation. It increases developer productivity.” The MetaEdit+ website also introduces ten examples of worldwide commercial companies working in different domains that are using MetaEdit+ in their work. These companies include NOKIA® which uses the tool for generating modelling languages in domains related to mobile phones. Panasonic®, SIEMENS® and Bloor® also use MetaEdit+ for generating specific modelling tools for different domains of interest. Safa, L., a MetaEdit+ user from Panasonic® commented that “Even as a beginner with MetaEdit+, I could define a domain-specific activity language in about six hours...”. It is believed that this gives an idea of the importance, existence and the use of meta-CASE tools in an industrial setting, reducing the effort and cost of specifying and generating domain specific modelling tools.

2.6 Meta-modelling

The target modelling language and its associated target CASE tool are specified through a meta-modelling process that includes describing the syntax of the language using a meta-model and its semantics using constraints. In all of the following discussion, specifying or defining the required modelling language means by default the specification of its associated CASE tool. **Meta-modelling** is the process of creating a model, the meta-model, for the required (target) modelling language by defining its syntax and semantics or “the act of creating a model of a modelling language, thus defining its abstract syntax and semantics” (Kirchner & Jung, 2007). Similarly, Clark, Evans, & Kent (2003) define the meta-modelling as an approach through which a language is defined by constructing a model of the abstract syntax of the required language. A supportive detailed meta-modelling process

definition has been introduced by Sommerville, Welland, & Beer (1987). They state that meta-modelling is the process of defining the vocabulary, syntax and semantics of a modelling language. *Vocabulary* includes the symbols and notations; *syntax* includes a description of the rules governing the connectivity and structure of a model; and finally, the *semantics* includes guidelines and rules that limit the user options and leads to a good design (model) and meaningful structures. Nordstrom, Sztipanovist, Karsai, & Ledeczi (1999) define it as “the process of creating other models”. Put simply, it is the process of modelling (or specifying) the required target domain specific language using the meta-model.

The ***meta-model*** is the model “used to specify a language”. It is the model that specifies the syntax and semantics of the required modelling language (Kleppe, 2009 page 68). In more detail Nordstrom, Sztipanovist, Karsai, & Ledeczi (1999) and Ledeczi, Maroti, & Volgyesi (2001) state that it is possible to differentiate between two parts in the meta-modelling process, the syntax definition and the semantics definition.

Modelling the syntax: Syntax and lexicon definition, is achieved through the meta-model itself by defining the modelling object types and the allowable relationship types between them (Nordstrom, Sztipanovist, Karsai, & Ledeczi, 1999). The definition of meta-modelling above by Kirchner & Jung (2007), Kleppe (2009 page 76) and Clark, Evans, & Kent (2003), differentiates between the abstract syntax and concrete syntax. The abstract syntax, for graph based languages, defines the node and edge types with their structure and properties. The abstract syntax is part of the meta-model. The concrete syntax of a language defines the graphical representations of the nodes, edges and their defined properties. The concrete syntax is not part of the meta-model; however, (Kirchner & Jung, 2007) introduces it as an important feature that meta-CASE tools should offer and allow to specify by connecting the concepts and types in the abstract syntax to graphical representations. (Nordstrom, Sztipanovist, Karsai, & Ledeczi, 1999) introduce that in the context of specifying a domain specific modelling language (editor) for “processor modelling” the syntax definition step includes defining the object types “processor” and “sensor” while the relationship type is represented by the relation “connectedTo”.

Modelling the semantics: The meta-model, in general, specifies the abstract syntax of the required language, however, it does not specify the legal constructs or the correct models in the target language or the meaningful structures, the language semantics. The semantics specification task is left for the *constraints* to perform (Ledeczi, Maroti, & Volgyesi, 2001). Ledeczi et al. (2001) differentiate between the definition of syntax and the definition of semantics and they insist that meta-models only describe the syntax of the required language but not the semantics. Clark, Evans, & Kent (2003) state that the meta-model may, optionally, include the concrete notation and semantics of the target language. Nordstrom, Sztipanovist, Karsai, & Ledeczi (1999) add that the second meta-modelling process part, semantics definition, is achieved by constraint specification and can be represented, in the processor modelling editor example introduced above, as the number of allowed individual processors to be connected to sensors using the “connectedTo” relationship (Nordstrom, Sztipanovist, Karsai, & Ledeczi, 1999). Some literature extends the work of constraints to be also required in the syntax definition task as Kirchner & Jung (2007) introduced. They stated that constraints can participate as a technique for specifying the abstract syntax for defining the language concepts’ purposes as in the constraint that prevents circular relations. They also agree with Ledeczi, Maroti, & Volgyesi (2001) that the easiest way for semantic specification is using constraints. The other alternative is to map the language concepts to a common programming language but this choice is not feasible because of its complexity and the need for specialised tool support of such mappings.

Nordstrom, Sztipanovist, Karsai, & Ledeczi (1999) differentiate between static semantics and dynamic semantics and they classify constraints based on that. Static semantics represents the constraints that are known before the target modelling language is generated and, consequently, they include all the constraints that can be defined at the meta-modelling time. The dynamic semantics includes constraints that specify the meaning of a concrete instance model of the generated modelling language in a specific context. These constraints cannot be defined at meta-modelling time because they are specific to the contexts of the model instances themselves, the thing that the meta-model has no prior knowledge about. As an example, the static semantics in a meta-model for Use Case diagram specify that “it is allowed to connect two Actors using a Generalisation edge”. However, the constraint “it is not allowed

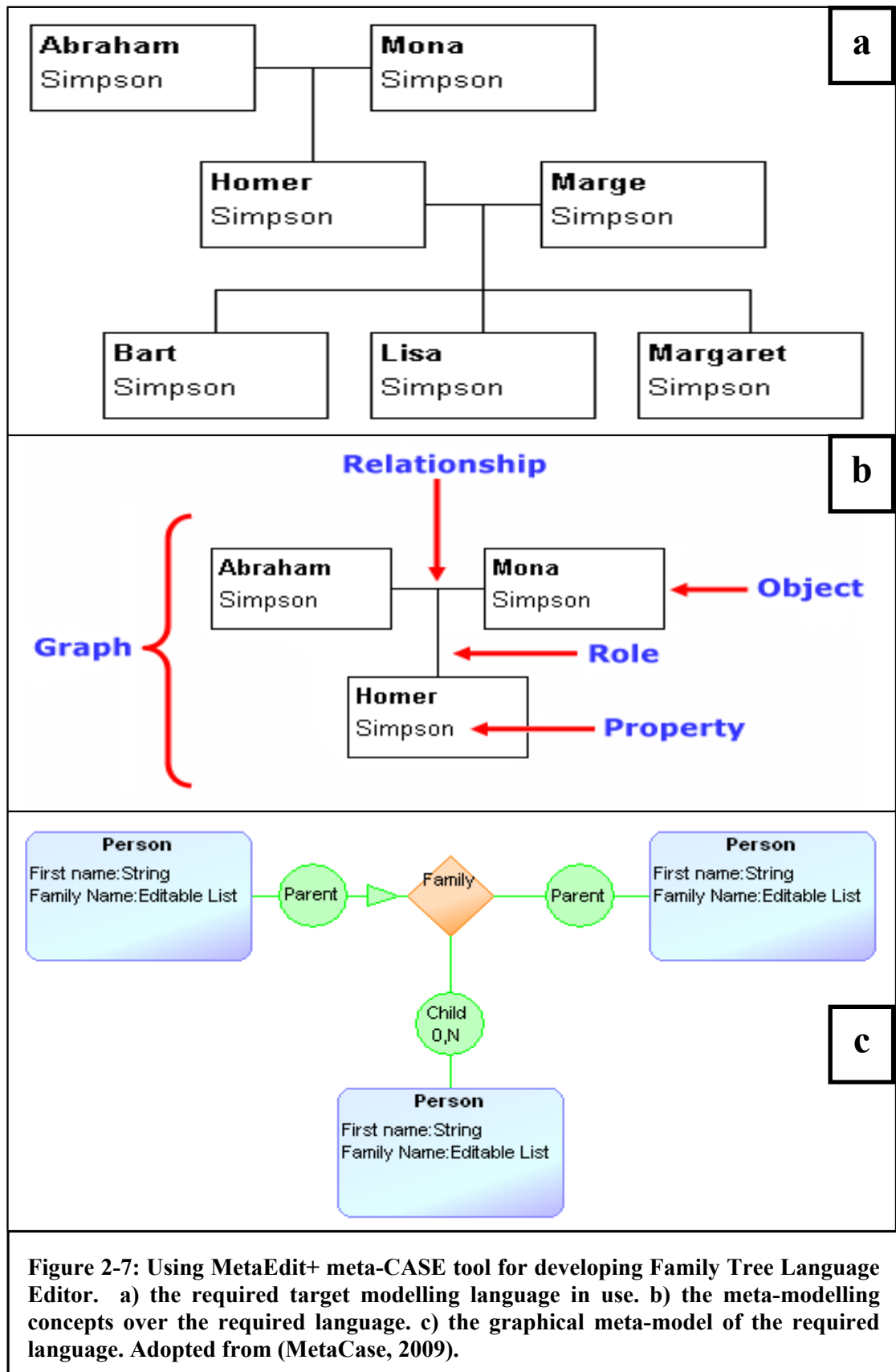
that the Administrator Actor generalises the User Actor” in an instance of the generated Use Case diagram that models the requirements of a database system. Such constraints cannot be specified by the meta-model because it is related to a specific context, the database system to be modelled. This research is focusing on the static semantics constraint specification rather than solving the problems of dynamic semantics realisation.

The meta-model is constructed using a *meta-modelling language* which according to (Ledeczi et al., 2001) is a predefined language that is rich enough to describe specific languages of different domains. Kirchner & Jung (2007) suggest using GPLs such as Unified Modelling Language (UML), Entity Relationship Model (ERM), or Event-driven Process Chains (EPC) as meta-modelling languages. Kirchner & Jung (2007) describe using the meta-modelling language as the “approach” of building the meta-model to define the syntax and semantics of the required language. They advocate using a graph-based meta-model approach for defining the visual modelling languages which is semi-formal but intuitive in building the meta-model. Another approach could be the grammar-based approach such as Extended Backus-Naur Form (EBNF) which is a formal and precise way of describing the required modelling language. Because the last is time consuming, the graph-based approach is preferred.

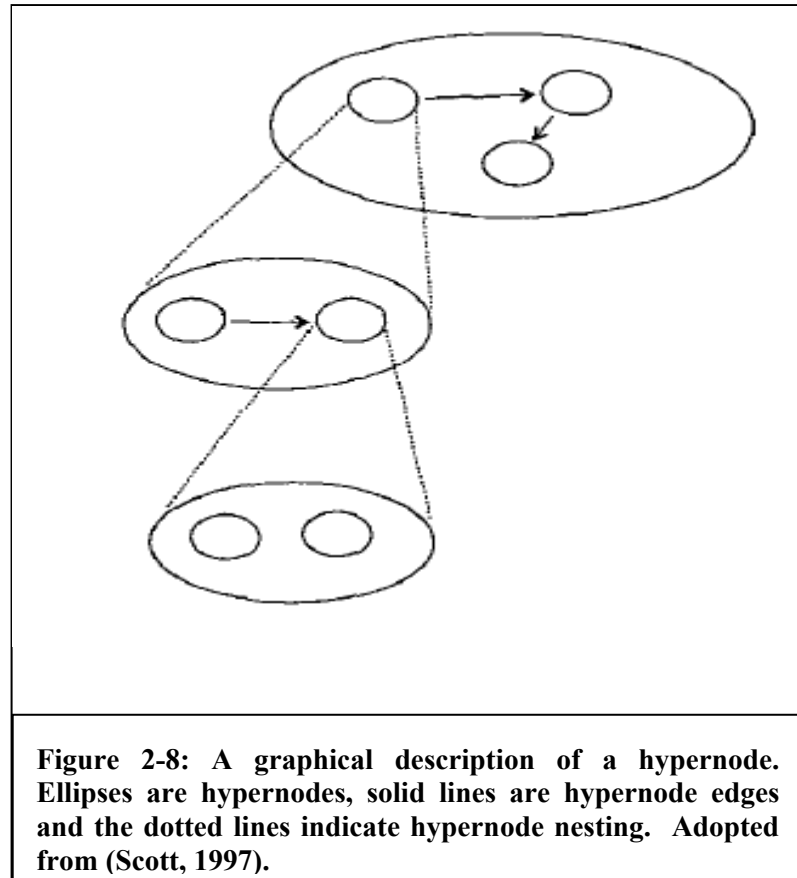
Different meta-CASE tools have different meta-modelling approaches with different concepts and depend on different techniques. Isazadeh & Lamb (1997) introduced a classification of meta-CASE tools based on the technique used to build and construct the meta-model. This classification includes three techniques, ER-diagram based, OO-based, and graph-based. They also gave examples such as “MetaView” as a meta-CASE tool that depends on the ER diagram. It allows a meta-model to be built based on EARA/GE (Entity-Aggregate-Relationship-Attribute with Graphical Extensions) concepts. These concepts are defined using a textual language called the Environment Definition Language (EDL). Isazadeh & Lamb (1997) document that to be able to complete the modelling target language definition there is a need to define constraints over the EARA objects. The constraint language Environment Constraint Language (ECL) is used for this purpose. The constraints are used to enforce completeness and consistency checking on the objects.

Toolbuilder is another meta-CASE tool that has an ER-based meta-model. It depends on generating diagram editors that are parameterised with the required data structure, interfaces, and the required symbols and graphics. These three parameters define the meta-model that captures the concepts of the required modelling language. However, Isazadeh & Lamb(1997) document that there is a need for constraints to be able to specify the behaviour of the editor. This is achieved using constraints that are defined using the C programming language.

“MetaEdit+” is a commercial OO-based meta-CASE tool. It depends on building a meta-model that captures the Graph, Object, Property, Port, Relationship and Role (GOPPRR) concepts. Figure 2-7-b shows some of these concepts in relation to the required target modelling language which is the Family Tree modelling language in this case. The (GOPPRR) is considered also as the meta-modelling language for this tool. The tool depends on a graphical meta-model which is shown for the Family Tree modelling language in Figure 2-7-c. The meta-model represents the abstract syntax as it shows the different vertex and edge types. Each object participating in the meta-model should be described using the above concepts which helps in capturing the required domain. Finally, MetaEdit+ uses a specific constraint language to define the semantics of the required language (Kelly, 2009).



Isazadeh & Lamb (1997) introduced examples of graph-based meta-modelling techniques such as in the meta-CASE tool “CASEMaker”. CASEMaker depends on hypernode graph to capture the concepts and models the required language Figure 2-8. The meta-CASE tool uses a hypernode scheme constraint definition language to specify the allowed structures.



From the above definitions and arguments, it is possible to conclude that meta-modelling is complex, needs time, and, in the case of many tools, depends on experts to do the specification (Borning, 1986; Draheim, Himsl, Jaborning, Leithner, Regner, & Wiesinger, 2009). Pohjonen (2005) documented part of the meta-modelling as needing a considerable amount of manual programming in meta-CASE tools to specify and define the target language and proposed reducing or eliminating the need for programming as a solution to reduce the difficulty of using meta-CASE tools. It is also possible to conclude from the above discussion that meta-modelling, as a modelling language specification process, can be subdivided into two main activities, the specification of the diagramming language elements and the specification of the constraints over these elements (Smith, Cypher, & Spohrer, 1994; Draheim, Himsl,

GME also depends on an OCL extended language called Embedded Constraint Language (ECL) (Gray, Bapty, Neema, & Tuck, 2001) over the meta-model to “specify the static semantics” of the target language, in which these constraints are enforced (Ledeczi, Maroti, & Volgyesi, 2001).

The meta-CASE tool KOGGE uses an extended version of entity relationship diagram (EER) to represent the concept of the target language and a constraint language called GRaph specification Language (GRAL) which is a Z-like formal language. The EER role in the meta-model describes the concept of the required domain specific modelling language using five different blocks to represent entity types, relationship types, attributes, generalisations and aggregations. GRAL is an assertion language that is used to specify constraints over the EER description (Ebert, Süttenbach, & Uhe, 1997).

2.7 Constraints

(Related to Study One, Chapter 5)

2.7.1 Importance of Constraints in Meta-modelling

Constraints are defined in the context of diagram editors as rules that govern and restrict the behaviour of the designer. They are considered as limitations to the available alternatives and signs to guide and enforce the behaviour of the designer towards producing good designs (Offen, 2000). They are rules that limit the available alternatives to achieve a task, and enforce conformity with a specific software design process methodology (Jankowski, 1997; Offen, 2000; Scott, Horvath, & Day, 2000; and Bergmann et al., 2010). Design constraints are an important means of evaluating the correctness (consistency) of a model (Groher, Reder, & Egyed, 2010). This research is interested in software engineering modelling language constraints (sometimes ‘design constraints’. Examples of this type of constraints are connectivity constraints and cardinality constraints such as “there must be a path between the Start State vertex and every other vertex in the State Transition Diagram” and “it is not allowed to have more than one End State in the State Transition Diagram”.

All the reviewed meta-CASE tools literature suggests that constraints play a vital role in the meta-modelling process and form one of the pillars that meta-CASE tools are based upon. The literature agrees that the semantics of the required

modelling language is specified by constraints. According to Kirchner & Jung (2007), specifying the semantics of a modelling language with its meaningful structures is complicated (cannot be expressed using the graphical meta-model alone) and can only be achieved using constraints. Some others add that the constraints also participate in the syntax definition. The meta-CASE tools presented above as examples of the meta-modelling process clearly show that all of them depend on constraints in the modelling language definition. In all of them constraints are participating and have a place in the definition process. Kirchner & Jung (2007) introduce constraints as a main feature that should exist in any meta-CASE tool to refine and complete the definition of required modelling language. All of this gives an idea about the importance of the constraints and their existence as a part of the modelling language specification process through meta-CASE tools.

Constraints were also introduced as a possible solution to increase the flexibility of meta-CASE tools by Gray & Welland (1999) through allowing customising and editing constraints in already generated diagram editors. The idea has been implemented in the meta-CASE tool GME (Karsai, Nordstrom, Ledeczi, & Sztipanovits, 2000). GME provides several techniques to facilitate incorporating constraints into meta-modelling which indicates the importance of constraints and highlights their role in meta-modelling. Karsai, Nordstrom, Ledeczi, & Sztipanovits (2000) even described depending on constraints for CASE tool specification through meta-modelling by the term “constraint-based meta-model”.

In GME defined constraints can be reused through calling them from other constraints or other functions. GME also allows constraints to be added, removed and evaluated on demand. It implements an interactive tool, the constraint browser, which enables the available constraints in the database to be browsed, presenting their definition state and attributes. The browser also allows any of them to be evaluated and disabled temporarily. Another implemented facility that is related to constraints in GME is the constraint debugger. This facility assists the CASE tool user to discover problems in constraint definitions. However, this facility also can be turned off when the user is confident of constraint definition correctness. This shows the role that constraints play in the domain of meta-CASE tools (GME, 2005). The following paragraph introduces some examples of using constraints in meta-CASE tools.

The meta-CASE tool GME uses the UML class diagram as a meta-model and OCL as a constraint language for modelling language semantics specification (Nordstrom, Sztipanovist, Karsai, & Ledeczi, 1999 and Ledeczi, Maroti, & Volgyesi, 2001). They specify the static semantics using constraints which are responsible for defining the legal and correct models in the target language. They document that if the meta-model fails to capture information of the target language for a reason like the inability to directly compile the captured information, constraints can perform the job. In such cases, GME uses the required constraints, in the form of OCL expressions, to overcome the limitations of the meta-model. Again here, the constraints appear as part of the meta-model by adding more information to it instead of being a separate part. They provide cases of such replacement of the meta-model with constraints as “multiplicity information of containment, membership and connection cardinality definitions”.

A similar problem of not being able to capture the required integrity conditions in the KOGGE meta-CASE tool was solved using the constraint language GRAL. KOGGE uses EER diagrams to model the required constraint language. The constraint language GRAL is considered as an important part of the meta-modelling adopted in the meta-CASE tool (EER/GRAL). This is because EER alone is not able to specify the required languages (Ebert, Süttenbach, & Uhe, 1997). This point was also agreed by De Lara & Vangheluwe (2002) who document that an ER diagram that is used as a meta-model in the meta-CASE tool ATOM3 must be extended with a constraint language (OCL or Python expressions) to be able to define the target language.

In a meta-CASE tool (called the ISI), Goldman & Balzer (1999) divided the meta-tool development process into two parts, graphical user interface generation and the part that provides the feedback such as the feedback of problems and design correctness, which work in the same manner as the constraints in other meta-tools. They document that specifying the target language “units”, vertices and edges, is not sufficient to produce meaningful designs. They call this process “graphical user interface specification” and they add that this specification requires only superficial knowledge of the domain to be modelled. In this case the user may use his/her deeper engineering understanding of the domain specific language to generate meaningful

designs or use the generated editor to construct artefacts without the deeper understanding. By contrast, the feedback part of the tool requires a deep understanding of the domain and they have achieved its specification through independent components, the analysers. These analysers are feedback programs and have exactly the same role as the constraint checker. In other words, they depend on the constraints and they claim that the constraints are more important than the other part for the diagram editor specification. This is because the constraints control the behaviour in the diagram and ensure design correctness.

This research focuses on the modelling language syntax and semantic related constraints. Examples of such constraints have been introduced by (Tolvanen, Pohjonen, & Kelly, 2007) as those define the legal connection types between object types, element occurrence and uniqueness. In this research, such constraints include ones like “Non-Terminal State vertices must have unique labels in a State Transition Diagram”.

2.7.2 Constraint Definition

The constraint definition process is complex, time-consuming and error-prone (Groher, Reder, & Egyed, 2010). Liu, Hosking, & Grundy (2007b) claim that a common difficulty of meta-CASE tools is specifying behaviour, including constraints. It is a fact that all meta-CASE tools depend on a constraint language to define the required constraints for the required CASE tool specification. Although some tools may not have a distinctive step of constraint definition (e.g., it is considered as part of the whole meta-modelling such as in MetaBuilder (Ferguson, Hunter, & Hardy, 2000)), many other tools adopt different constraint definition techniques to achieve this task. Some literature has documented the problem of the difficult of constraint definition and has proposed different solutions and techniques to handle it. Part of the solution was to change the technique of constraint definition such as using form-filling or a visual language instead of depending on direct constraint editing using a scripting language.

What has been noticed in this field is that the documentation is very limited. Literature that documents meta-CASE tools provide very little space documenting the constraint definition technique in the meta-CASE tool compared to the space given

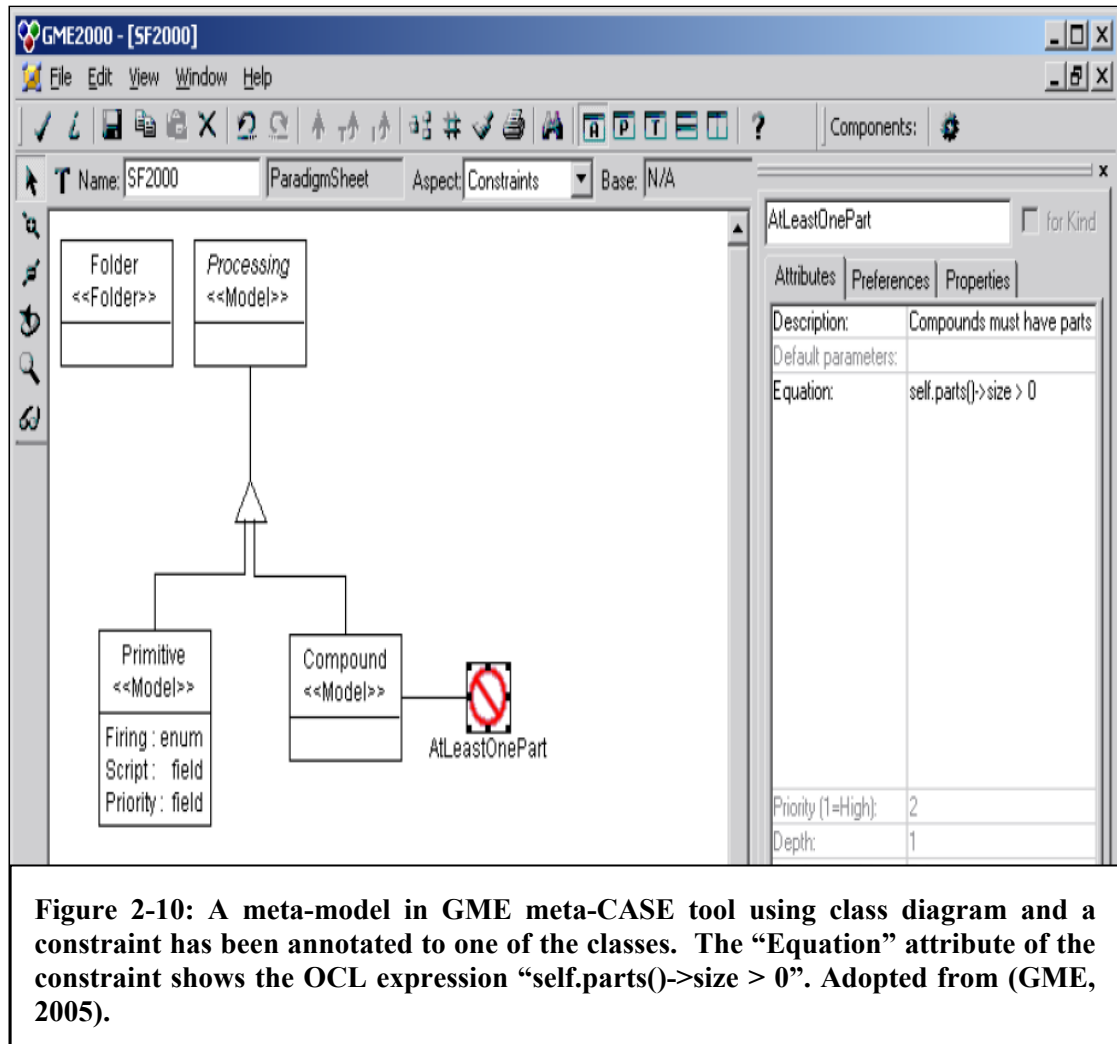
for documenting the meta-model. In general, a constraint language should be used to define constraints in meta-CASE tools as has been introduced in some examples in the previous sections. However, what this research is interested in is the technique used for constraint specification. The following subsections introduce different techniques that are used in constraint specification in meta-CASE tools. There is also a documentation of the attempts of handling the problems of one technique by using another.

2.7.2.1 Formal Constraint Language

Using formal language for constraint specification in meta-CASE tools is the oldest, and one of the most common, techniques used in meta-CASE tools. In such tools, constraints are specified using direct constraint language editing through an editor. Examples of these tools are “Metaview” that uses Environment Constraint Language (ECL) (Findeisen, 1994), and “MaramaTatau” (Li, Hosking, & Grundy, 2009). Some modern tools use OCL as a formal language such as in “MaramaTatau” and Graphical Modelling Framework (GMF) which depends on Eclipse Modelling Framework (EMF) for providing the meta-model. However, tools that use OCL should, by default, use a UML-based or closely related meta-model which is usually a Class diagram such as in GMF or Extended Entity Relationship (EER) as in the case of MaramaTatau (Li, Hosking, & Grundy, 2009).

As a consequence of using a formal constraint language and using the direct editing of the language as a technique for constraint specification, the users of such tools must learn the constraint language programming to accomplish the job. This is considered as a major problem with this technique. Moreover, in such tools the mapping of the designer’s empirical understanding and experience of the domain specification to an abstract textual representation is considered another problem in this technique (Bimbo & Vicario, 1995). The same problem has been noted by Liu, Hosking, & Grundy (2007b). They stated that using OCL causes the problem of a wide gap and separation between the visual specification of the meta-model and the textual specification of the constraints. They document that GME tries to bridge the gap by annotating the visual meta-model elements to indicate the application of constraints, but these constraints are still hidden. Figure 2-10 shows a meta-model of a class diagram in the GME meta-CASE tool. The “Equation” attribute of the

constraint (on the left hand side) shows the OCL expression “self.parts()->size > 0” which indicates that at least one part of the annotated object should exist in the diagram.



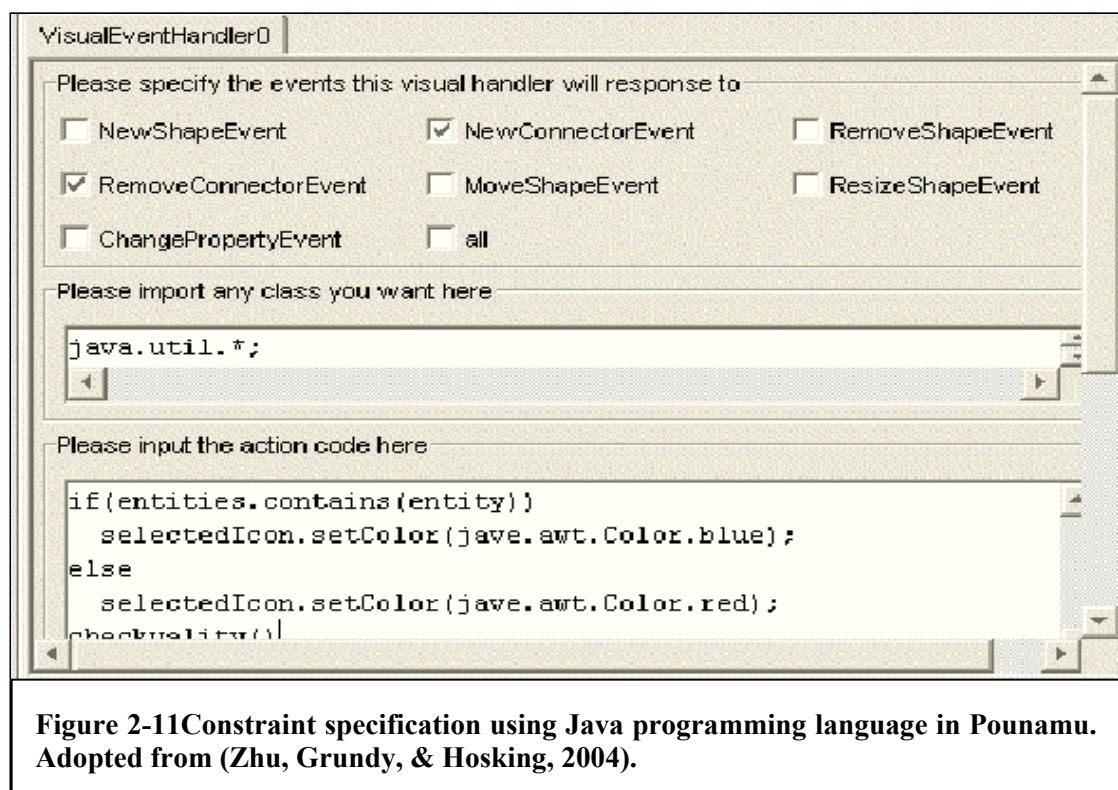
Some tools tried to reduce the difficulty of programming formal constraint languages through providing supportive editors. As an example, MaramaTatau uses a novel editor that provides a view of the constraint list juxtaposed to the meta-model. The tool also allows constraint navigation and provides an OCL constraint formula debug viewer which allows checking the OCL expression correctness.

The meta-CASE tool KOGGE (Ebert, Süttenbach, & Uhe, 1997) uses GRAL as a constraint language and considers it as a part of the meta-model (EER/GRAL). GRAL is a Z-like assertion language that adds information to the diagram to specify the integrity conditions. It specifies constraints on the values of the attributes of

vertices and edges. These constraints include the existence of a certain path in the graph and cardinality restrictions of vertices and edges. They are defined as a set of predicates which refer to the EER description and defined using a textual editor.

2.7.2.2 Text-based Constraint Specification Language

Text-based constraint languages are those languages that are used to define constraints using direct script editing but they are not formal constraint languages (discussed in the previous section). Zhu, Grundy, & Hosking (2004) introduced the problem of constraint specification difficulty in traditional formal constraint programming languages and proposed the solution of another form of text-based programming approach. They introduced the Java programming language for event handling and behavioural constraint specification as a replacement in the meta-tool “Pounamu”.



In “Pounamu” a Java program is written by the user to specify a specific behaviour, including constraints. This is done by providing the user with a Java code editor from within the meta-CASE tool as shown in Figure 2-11.

The same problem was addressed and the same “escape to code” solution has been introduced by White & Schmidt (2005) in the meta-tool Generic Eclipse Modelling System (GEMS), the VisualStudio DSL tool (Liu, Hosking, & Grundy, 2007b) and JView (Grundy, Mugridge, & Hosking, 1998a) for event handling specification. However, in an evaluation for their meta-tool “Pounamu”, Liu, Hosking, & Grundy (2007a) have reported that users dislike the event handler specification tool because Java is a prerequisite of using it. This was introduced in Liu, Hosking, & Grundy, (2007b) as a problem that requires repetitive coding and knowledge of the meta-tool API. ATOM3 provides options to choose the technique for constraint specification; it is possible to use OCL as a constraint language or Python expressions. In both cases, the constraints must be edited by the user (De Lara & Vangheluwe, 2002). Similarly, Goldman & Balzer (1999) used programs written in C++ or Visual Basic called ‘analysers’. These programs are independent from the meta-tool and can be predefined which allows them to be used when required. They provide feedback in the generated editor based on a request of the designer.

It is believed that this category of constraint specification technique suffers from similar problems that exist in the formal constraint language category, maybe with a lesser severity. This is because text-based constraint specification even with Java code still requires the user to know Java programming which is the problem addressed in “Pounamu”. However, it is also believed that Java as a general programming language is much more common than formal constraint languages. Moreover, the problem of the gap between the text of specification and the visual form of application still exists because constraints are specified in Java through text editing.

Microsoft Visio® is another tool that can work as a meta-CASE tool and in which rules can be specified (Microsoft, 2010). It is possible to manipulate the GUI of Visio to construct templates using the object ShapeSheets which works under the developer mode. The user of Visio can specify constraints such as preventing connections between specific vertex types using a Visual Basic program. This is called diagram validation (Microsoft, 2010). This is also considered as another application of the “escape to programming” technique for constraint specification.

2.7.2.3 Visual Programming Language

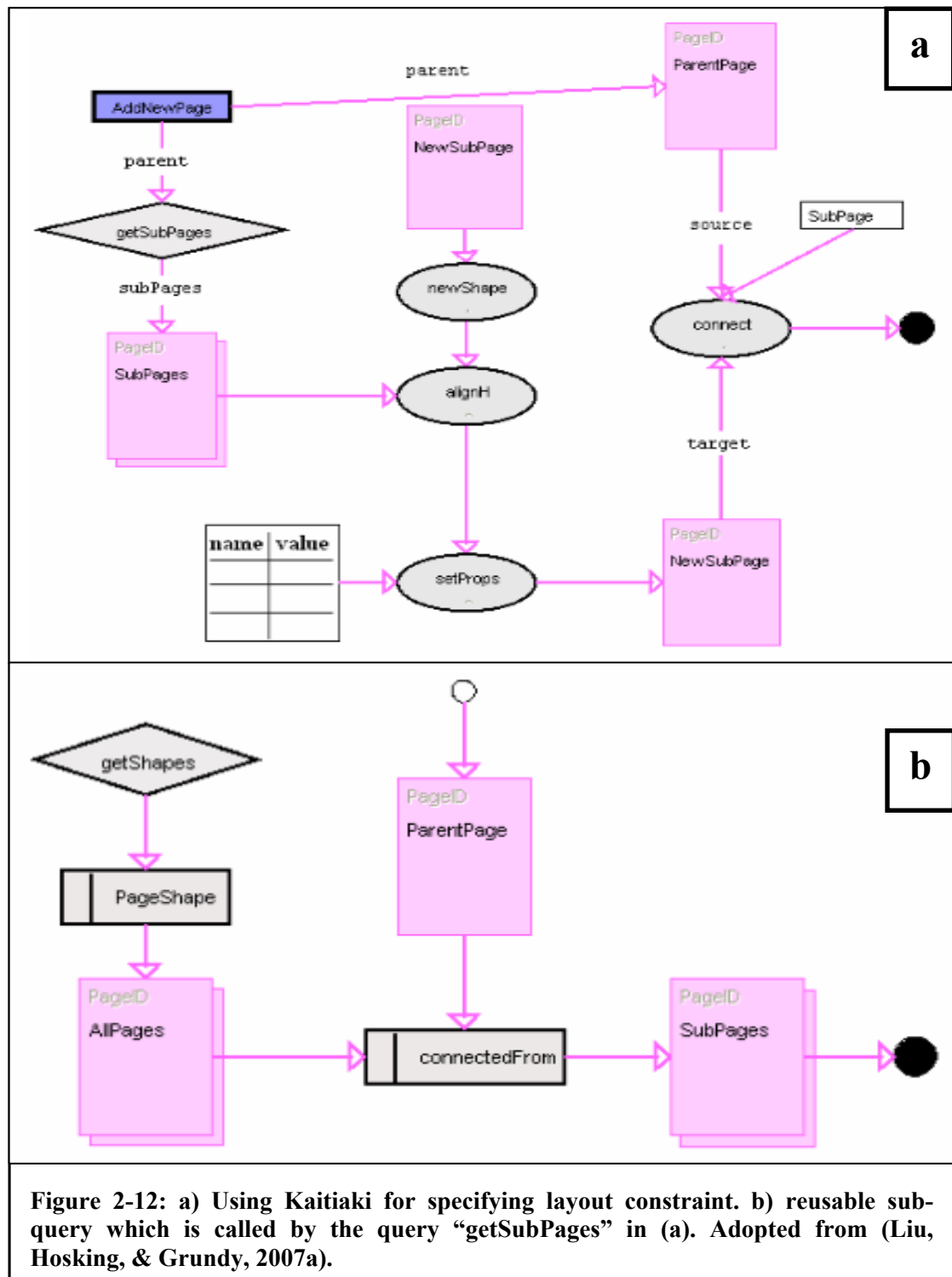
Because of the difficulty of Java-based constraint specification and as an enhancement, Liu, Hosking, & Grundy (2007a) have invented a visual programming language approach for event handling specification purposes. The visual language, “Kaitiaki” replaced the Java scripting approach in a newer version of “Pounamu”. Kaitiaki is a visual event handler that uses the dataflow metaphor, called Event-Query-Filter-Action (EQFA) to specify event handling in “Pounamu”. The language depends on a set of visual concepts called building blocks such as filters, iteration, query and start and end points of the data-flow diagram that describes the event and the required behaviour, the action. The flow of data is described visually through arrow shape connectors and data propagation links. The aim of this approach was to simplify the behavioural constraint specification especially for non-programmers. Figure 2-12-a and Figure 2-12-b show the visual language that is used to specify the events. Figures show how to specify a layout constraint when the event of adding a new page in a website occurs. In Figure 2-12-a the query “getSubPages” calls a sub-query which is considered as a reusable package that is shown in Figure 2-12-b.

Similarly Henkel & Stirna (2010) in the business modeller tool “Medix” introduced “microflows”, a visual programming technique to handle events. Microflows are structures of process models that describe events triggered by actions. They use notation extended from UML activity diagrams to construct the models. The events that trigger microflows can be set on objects (or concepts) in the meta-model, such as “whenever a Complaint is updated or deleted”³. Events also can be made to start microflows when triggered by GUI forms such as pressing on a button.

Microflows are flexible because they can use loops to express behaviour and they can call other microflows, the feature that has also been introduced in “Pounamu”. Microflows are not fixed such as in case of the “Pounamu” language;

³ This constraint refers to a business process model; hence, the reference to a (client) “complaint”.

instead, they are extensible by calling Java code procedures and external web services.



JViews, which is a toolkit that extends JavaBean API, defines “event handling” through procedures of a programming language (Grundy, Mugridge, &

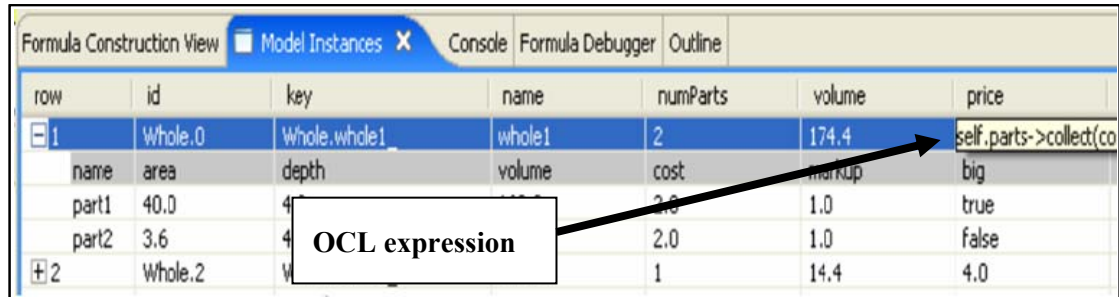
Hosking, 1998a). It uses a visual language called Architecture Description Language (ADL) for attaching annotations (in label format) to indicate the specified events on the relations between different components.

In Grundy, Mugridge, & Hosking (1998b) they describe a tool, JComposer that specifies and generates multi-view visual environments. The meta-tool specifies behaviour including constraints through an event handler. A visual event-flow language is used to specify the required language semantics. The language is similar to the one introduced in Liu, Hosking, & Grundy, (2007a) and it has almost the same concepts and visual constructs, such as filters. JComposer uses another meta-tool that specifies visual constraints which is BuildByWire (Mugridge, Hosking, & Grundy, 1998). The tool is used for specifying components and objects' visual representations. BuildByWire is also used for component visual specification in the tool JViews. Although both BuildByWire and Kaitiaki are easier to use than text-based specification (using Java code in this case) and solve the constraint specification complexity problem, both suffer from a significant disadvantage which is the lack of a user interface required to capture the model level specifications and "model level constraints" (Liu, Hosking, & Grundy, 2007b). They proposed using a spreadsheet-like user interface for constraint specification to solve this problem.

2.7.2.4 Spreadsheets

Liu, Hosking, & Grundy (2007b) present spreadsheets as a solution for the constraint specification difficulty problem. It is claimed that the technique has not been used in the domain of meta-CASE tools before Liu, Hosking, & Grundy (2007b) used it in MaramaTatau. However, the idea of using spreadsheets for constraint specification has been proposed previously in Burnett, Atwood, Djang, Gottfried, Reichwein, & Yang (2001) and Engels & Erwig (2005). In "ClassSheets", Engels & Erwig (2005) presented a tool that manages transforming a class diagram specification to spreadsheets. Liu, Hosking, & Grundy (2007b) applied a spreadsheet-like interface for constraint specification in the MaramaTatau meta-CASE tool. The spreadsheet interface was used for easier definition of the property-change event handler constraints. Because MaramaTatau uses OCL as a specification language, spreadsheet cells are filled with OCL expressions (Figure 2-13). This figure shows the GUI of the spreadsheet that specifies an aggregation constraint. The GUI shows

different properties for the “Whole” object as can be seen in the figure. The property “price” is calculated using a formula that is specified using OCL. In this specific example, the formula says that the “Whole” object price is calculated through some calculation that is applied over each of its parts objects (here part1 and part2). However, the spreadsheet technique has not been used in any other meta-CASE tool.



row	id	key	name	numParts	volume	price
1	Whole.0	Whole.whole1	whole1	2	174.4	self.parts->collect(co
	name	area	depth	volume	cost	markup
	part1	40.0		2.0	1.0	true
	part2	3.6		2.0	1.0	false
2	Whole.2			1	14.4	4.0

Figure 2-13: Spreadsheet-like GUI (adapted from Liu, Hosking, & Grundy(2007b))

2.7.2.5 Form-Filling

This technique is another attempt to escape from the direct editing of constraints for specification purposes. It depends on editing form fields which represent the properties of the required constraint. Form-filling, which can exist in different formats such as wizards, has been implemented in the commercial meta-CASE tool MetaEdit+ (Kelly, 2009).

Figure 2-14-a and Figure 2-14-b show specifying a cardinality constraint and a connectivity constraint respectively using the form-filling technique available in MetaEdit+. It has been commented by Kelly & Tolvanen, (2000b) that MetaEdit+ is “one of the most widely known and used metaCASE tools” and is used by Nokia designers for developing their own methods. Bock (2007) also stated that MetaEdit+ has proven it is powerful for capturing domain concepts. The constraints in MetaEdit+ are merged within its meta-model, GOPPRR. For example, cardinality constraints can be specified through the Role concept in the meta-model. In general, specification of such constraints is conducted through wizards as Liu, Hosking, & Grundy (2007b) introduced. The values are filled in to give the values of the properties such as the maximum and minimum cardinalities of a specific object or a specific relation.

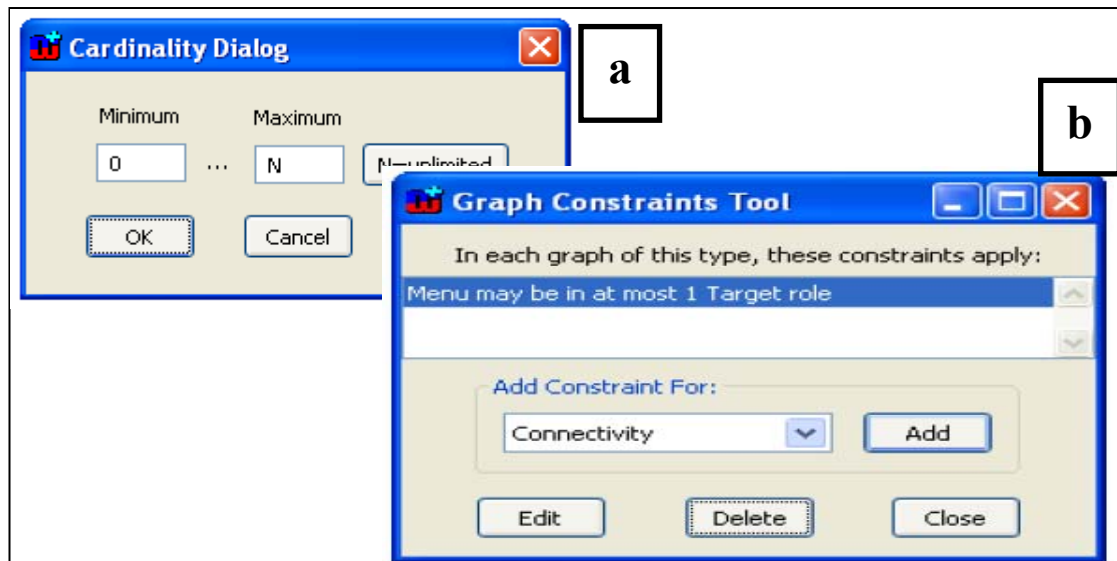


Figure 2-14: Form-filling technique in MetaEdit+. a) cardinality constraint specification. b) connectivity constraint specification. Adopted from (MetaCase, 2009).

The form-filling technique is also used in the meta-CASE tool “MetaBuilder” that uses hypernodes as a meta-model (Scott, 1997). The tool provides a form that asks the user to fill values for different properties. They call this form a “constraint dialogue”. Figure 2-15 shows the constraint dialog in MetaBuilder for specifying a cardinality constraint.

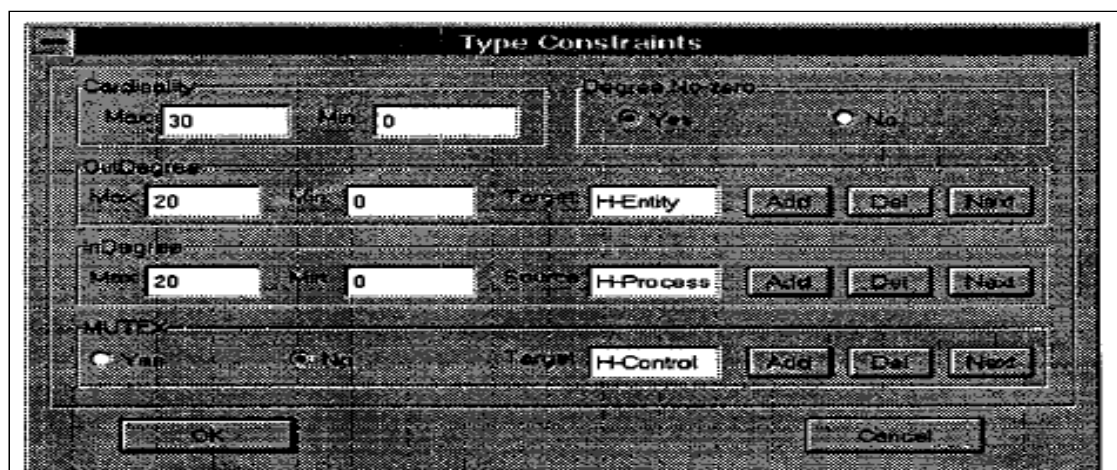


Figure 2-15: Specifying a cardinality constraint using form-filling technique in MetaBuilder meta-CASE tool. Adopted from (Gong, Scott, & Offen, 1997).

In another meta-CASE tool, “CASEMaker” that also depends on hypernodes as a meta-model, it is believed that the same technique is used for constraint definition

because it is believed that both are the same tool but with different versions. This is not documented explicitly; however, it can be inferred from the screenshots⁴. It is believed that the spreadsheet technique can be considered as a form-filling technique but with a spreadsheet-like GUI because it appears that the user requires to fill the cells with OCL expressions. The way the spreadsheet is implemented in MaramaTatau supports this idea as the objects are connected to the constraints using the spreadsheet but the constraints specification was done using OCL expressions inserted in the cells of the spreadsheet-like GUI (Figure 2-13).

A similar technique has been used but in, a wizard GUI. The user fills the properties of the required constraints which are specified using OCL in the graphical modelling tool IBM Rational Software Architect (RSA) (Wahler, Koehler, & Brucker, 2006). The tool depends on some constraint patterns and the user chooses from these patterns then edits and modifies the predefined constraint. The editing is organised through the wizard. Goldman & Balzer (1999) state also that their tool depends on a properties form for each of the specified domain elements (vertex or edge). Two of the properties specify the upper and lower bound numbers of each element.

2.7.3 Summary of Meta-CASE tools

Table 2-1 presents a classification summary of the meta-CASE tools based on the technique used in constraint definition regardless of the constraint language used. This classification, or its equivalent, has not been produced previously by others.

A potential candidate technique for constraint definition is Programming by Example (PBE). However, the discussion above and Table 2-1 show that this technique has not been used before for constraint definition in the context of meta-CASE tools. As a result, part of the novelty of this research lies in the fact that it focuses on the development and use of a PBE technique in the context of meta-CASE tool domain for modelling constraint specification.

⁴ A request of the CASEMaker meta-CASE tool was sent to the developers for evaluation purposes and the reply confirms that the tool no longer exists (Brooks, 2008).

Table 2-1: Meta-CASE tools classification based on the constraint specification technique.

Technique	Example	Technique Implementation	Author(s)
Formal constraint language	Marama	Object Constraint Language (OCL).	(Grundy, Hosking, Huh, & Li, 2008)
Text-based other than constraint language	Pounamu	Writing a Java program to implement the behaviour.	(Zhu, Grundy, & Hosking, 2004)
Visual language	Pounamu	Visual language each component of it has a predefined procedure.	(Li, Hosking, & Grundy, 2009)
Spreadsheet	MaramaTatau	Filling a spreadsheet-like interface with OCL expressions.	Liu, Hosking, & Grundy (2007)
Form-filling	MetaEdit+	Rule forms and templates.	(Tolvanen, Pohjonen, & Kelly, 2007)

2.8 Programming by Example (Related to Study One, Chapter 5)

Programming by Example (PBE) or Programming by Demonstration (PBD) is a technique that depends on introducing examples of data and values to the system which generalises the example and generates a program (a script) (Myers, 1993). This

technique was invented originally to make programming an easier task and available for non-programmers. The claim behind its invention is that, as the user knows how to perform a task and how the system should perform, this should be sufficient to create the required program without the need to have programming skills. In addition, it is claimed that visual programming is easier to understand for humans than textually based programming. Another motivation for using programming by example is that people are more capable of providing examples of what they want or mean than expressing this using a textual program (Myers, 1986). Cypher (1993) introduces inferring the user intent from the example as the main challenge in applying such a technique.

According to Cypher (1993), the first programming by demonstration system is PYGMALION introduced in 1975. To avoid the confusion of systems with different techniques, Myers (1986) introduced a taxonomy to differentiate between the terms visual programming, program visualisation, and programming by example. Visual programming includes the visual programming languages, such as Class Diagrams and flow charts that can be interpreted later as programs. Program visualisation is not a technique to reduce the complexity of programming, however; it is a way to illustrate and explain a program that is already written using the textual conventional way using a programming language. This includes the illustrations of the program through animations or snapshots. The term ‘Programming by Example’ has been used to describe a wide variety of systems that use different techniques and depend on different concepts. However, Myers (1986) differentiates between two terms that describe two different system techniques, “automatic programming” and “programming with example”.

“Automatic programming” describes systems that infer from several examples through an algorithm. Accordingly they capture from the example the meaning of the user. “Programming with example” describes systems that only remember the example the user introduces and generate a program based on this example as an input. No inference is involved in such systems. Such a system executes the user commands and generates a program that allows reusing these commands and does exactly what the user did in the first place. Macros are the most common form of Programming with Example. Myers (1986) uses the term “programming by example”

to include both system techniques together. He represents the system that applies programming by example as the intelligent pupil that infers or intuitively abstracts and generalises the examples provided by his/her teacher.

2.8.1 Programming by Example Contexts and Tools

The Programming by Example technique has been introduced into several different contexts. The first to be documented here is the most relevant, the constraint definition. The closest example to the work presented in this dissertation is the programming by example tool “Peridot” (Myers, 1993). Peridot uses a programming by example technique to define layout constraints for graphical user interface components such as buttons and checkboxes. A Peridot user creates the GUI components and the visual effect on the GUI component responding to the action of the mouse without conventional programming. It depends on the user to introduce examples of the required constraint and the system infers the user intention from the example. The system shows the inference to the user to confirm that the inferred constraint is the required one. The system also uses a generalisation feature that allows the system to infer the intention of the user when repeated related actions are detected. The inference mechanism in Peridot is condition-action rules. The condition part of the rule specifies if the rule will be triggered or not; if true, then the action part is executed. The action part is composed of the required procedure to generate the required program for the constraint; this requires user confirmation of an attached English message first, however. Peridot depends in its work on 60 Lisp implemented rules that have four purposes:

- Inferring graphical constraints that specify the special relation of GUI components to each other. This rule type creates constraints between different GUI components such as the alignment of two buttons to each other.
- Generalising to infer the possibility of control structures, such as iteration, from repeated related actions. Such inferences are achieved straightforwardly such as in iteration which is inferred when the first two elements of a list are used.
- Inferring how the control structures can be created and represented using GUI components. This requires determining that some variables should be constants such as the y-axis or x-axis values depending on the GUI to be created. Such

inference is related to the previous one as the system should generalise and continue creating the rest of the GUI components, in a list as an example.

- Inferring the mouse effect on the GUI components. This type of rules infers when and how the GUI components should be changed and where the mouse should be to affect a specific component as a result of a mouse click or movement.

Myers (1993) justifies the use of the message to confirm the inference and applies the action part of the rule based on the fact that previous systems either ask the user to specify the constraints explicitly, which makes the systems difficult to use, or they infer the constraints automatically without confirmation from the user, which opens the door for many wrong inferences and guesses.

In the context of constraint definition by example, “Chimera” (Kurlander & Feiner, 1993) is a system that applies PBE to define geometric constraints to specify, edit and manipulate graphical objects and the constructions of user interface widgets. Chimera defines constraints based on multiple examples that are introduced one by one and for each a snapshot is taken. In each successive example, the previous example is edited and the snapshot is taken for the new edited example. The system then takes all the snapshots as input and infers the constraints that satisfy all the snapshots and applies them to the scene objects. Chimera defines geometric constraints that must be satisfied all the time after they are applied on the user interface objects. When one constraint is violated, because of a modification of the geometric position of an object, the system modifies the other objects to return the scene into a stable state that satisfies all the geometric constraints applied to it. Using this technique, the behaviour and the interaction of the user interface object is defined.

Demonstrational Rapid User Interface Development (Druid) (Singh, Kok, & Ngan, 1990) is a user interface management system that is similar in its work to Peridot. The difference between both systems is that the GUI components in Druid are predefined in a high level instead of low level as in Peridot. Druid deals with the complexity of wrong constraint guesses (inferences) by allowing the user to use a direct specification facility which depends on the form-filling technique to set the different required attributes to specify the desired constraint. This facility is used

when the system fails to infer the required constraint because of missing the rule that is required to infer the desired constraint from a specific example.

aCAPpella is a programming by example dependent system that is designed to help end-users to build context-aware applications, such as smart home systems involving sensor input to determine context dependent operations (e.g., “play my music from speakers that are near me”) (Dey, Hamid, Beckmann, Li, & Hsu, 2004). It uses a machine learning technique supported with user input to build the required application and generates its program. The aCAPpella system records the context of the example by capturing all the available data using all its available sensors. All the behaviours and actions during the recording are recorded to be presented by the user at the end of the data capturing (recording) session. The user reviews what has been recorded and annotates the data that is relevant to the behaviour that is required to be detected such as the actions that aCAPpella is required to perform on behalf of the user when specific events are detected. The user also specifies the start time and the end time of the indicated events and behaviours. This annotated data is entered into the learning system in aCAPpella for system training purposes. The same process is repeated several times so the system can learn to recognise the required important events. Later on, the system will be able to detect the events that trigger specific actions the user already specified to be conducted on his/her behalf. At this point the user asks to generate the program that performs the required actions when some specific events are detected.

Although the above systems involved the user as a collaborator/participant in the work of programming by example, some other systems introduced PBE as a way to avoid the involvement of the user in the work of the system. Gamut is such a system that uses a programming by example technique to help non-programmers to generate games (McDaniel & Myers, 1999). Gamut uses algorithms to infer from the examples without the intervention of the user to correct or redirect these inferences directly. However, Gamut allows the user to correct the wrong or partially wrong inferences indirectly through some techniques. “Hints” is one of the techniques to indicate the important objects to construct relationships between them. Another technique is through providing extra examples which allows the system to update and refine the generated code and the behaviour according to the newly introduced

examples. This is facilitated through the demonstration technique “nudges” which depends on providing the concepts “Do something” and “Stop that” which allow giving the system directions through positive and negative examples. Gamut also uses some other concepts to help achieving its work such as “guide objects” which are objects that are used to guide the inferences and the control of the program construction. These objects are visible only at program construction time and not in the generated program (the game). The behaviour in Gamut is represented by an “event” or a “stimulus” such as pushing a button and “action” or “response” which is the desired behaviour. Using the concept of the “timer” which specifies stimulus or responses related to the time is a distinctive feature of Gamut that has not been noticed in any other reviewed PBE system. As an example, the user can specify the movement of an object to be started after 10 seconds of a specific stimulus. According to this description of the Gamut system, it can be concluded that it depends on algorithms and concepts which require system-user interaction at a higher level than other PBE systems.

Another system that is also used to specify games using programming by example is Kidsim (Smith, Cypher, & Spohrer, 1994). Kidsim handles the problem of generalisation and abstraction by combining a graphical rule rewriting technique with PBD to specify the required game. The rule rewriting technique used here is similar in principle to the snapshots technique used in “Chimera” (Kurlander & Feiner, 1993). However, Kidsim involves the user interaction with the system to specify the important objects so the system can generate the required behaviour. The involvement appears when the user specifies the pictures and determines which one represents the “before” object and which represents the “after” object. The generalisation problem is also solved through user involvement by specifying the required level of abstraction using a pop-up menu to choose from all the available possible generalisations. For example, consider the case of a monkey object being required to jump over a rock; when the user demonstrates this to the system, the system should know the required generalisation for the rock object and the alternatives would be the rock or any object.

DocWizard is a system that learns through programming by example in the context of authoring and documentation (Prabaker, Bergman, & Castelli, 2006).

DocWizard developed to adapt for its work a technique called “follow-me”. This technique depends on the user interacting with an Eclipse framework GUI by performing a task and the system recording the steps that the user is performing. This generates a documentation of a number of steps called a procedure. A new user can playback the procedure and follow the documentation step by step supported by the system guide. The procedure is updated each time a new user performs different steps from the original procedure. This is called “incremental update”. A similar system is SMARTedit (Lau, Wolfman, Domingos, & Weld, 2001) which applies programming by example in the context of text editing. Another similar programming by example system that captures the interaction of the users with the GUI is CHINLE. This system watches the user performing a task using the interface of an application and generates a program to automate the task. It also generates the steps of interaction which allows the user to correct errors caused by the system generalisation (Chen & Weld, 2008).

The PBE technique has also been used in mapping between a state and its associated actions in robotic applications (Argall, Chernova, Veloso, & Browning, 2009). It has also been applied to solve the problem of extracting the constraints from a demonstrated task. This leads to generate information that is passed to a learning algorithm which helps teaching robots some tasks using examples (Calinon & Billard, 2008). PBE has also been used in the context of spreadsheets. Abraham & Erwing (2006) developed the spreadsheet system (Gencel) which uses the Visual Template Specification Language (ViTSL) to model spreadsheet templates. Using their software and its visual language, they designed an inference system that infers templates from example spreadsheets. This allows the flexibility of redefining the spreadsheets when the requirements changes and automating the repetitive tasks. Another domain of PBE application is web sites. Toomim, et al (2009) and Nichols & Lau (2008) introduced the use of PBE to enhance the user interface and to create mobile versions of web sites based on examples of user web site selections.

2.9 Example Polarity **(Related to Studies 2 & 3, Chapter 6)**

Recalling the definition of PBE, it can be concluded that the technique, depends mainly on introducing examples and the system generating a program based

on these examples. In PBE, the user introduces the examples to express the required program. In most of the above reviewed PBE systems, the examples are introduced to express a required behaviour. Although PBE itself and all the reviewed systems, consider the example as the first step of the technique, surprisingly, very little research attention has been given to the polarity of the examples, as no research has been conducted to evaluate this feature in separation. Examples introduced to any system can be either positive or negative. A positive example is one that expresses required behaviour while a negative example is one that expresses behaviour that is not desired.

Some PBE systems have introduced the notion of example polarity in different ways and implicitly without using the term ‘example polarity’. Myers (1993) documented that Peridot depends mainly on positive examples to express the required behaviour of the graphical interfaces; however, some constraints in limited cases need to be expressed using “negative examples”. He defines the negative examples as showing the system what not to do and he describes the situation of needing this type of examples as “exceptions”. Most of the PBE systems that report using positive and negative examples depend on those two example types together to refine the specification. In Gamut (McDaniel & Myers, 1999), negative examples are used to exclude behaviour from a generalised one. Gamut is an example of a complicated PBE system because it depends on many concepts and depends on intensive interaction between the system and the user to achieve the task as introduced above. One of the concepts is “nudges” which is considered as a valuable feature of the system that helps in refining the behaviour and the generated program code by introducing more examples. The “nudges” feature depends on introducing positive examples to tell the system what to do or to express the “Do something” concept and to introduce extra examples to refine the defined behaviour using examples that express the “stop that” concept. The last are considered negative examples that give the system direction in building the program.

MetaMouse (Myers, McDaniel, & Wolber, 2000) is another system that uses implicit negative examples to refine behaviour through conditional branches in the code. This system introduced positive and negative examples using the same concept as refinement for each other but in a different perspective. The user introduces

examples and all the examples are considered positive. The system depends on learning from the user's repeated examples. When a learned repeated action is detected, the system asks for user confirmation and shows the inference of the repetition. If the user rejects the inference, this is considered as a "negative example"; the generated code based on the positive examples is refined by building branches in the code. Another system that documents the example polarity concept is the InferenceBear or Grizzly Bear system (it has two names) (Frank, 1995). This system uses positive and negative examples explicitly to refine the behaviour of each other. These examples are used to generate functions that support and define GUI behaviour. Myers, McDaniel, & Wolber, (2000) add that "without negative examples, a system cannot infer many behaviours, including those using a Boolean-OR". Based on Myers, McDaniel, & Wolber (2000), this feature, the positive and negative examples, has been tested by some subjects and they found it difficult to use.

Heffernan (2003) proposed a tutoring system that can learn and includes inferring the structure of a human task from several example performances of the task. The system presented in Koedinger, Alevan, & Heffeman (2003) uses both positive and negative examples to help the computer infer the intent of a human. In addition, to be able use his system, a teacher introduces examples of how a problem should be solved (the task) and, in a separate process, a programmer continues the work by generalising the examples. This dependence on the programmer to generalise is a known limitation of the system identified by Heffernan that he was proposing to solve by generalising examples automatically.

In contrast to the research reported above, Hudson & Hsi (1993) are against using different polarities in examples. They criticise the way that different example polarities, or "counter examples" as they call it, work because that will add considerable work on the user's side. Instead, they recommend involving the user in a much simpler process which is selecting from alternative solutions.

2.10 Rules Augmentation and Learning (Related to Study Four, Chapter 7)

Programming by Example requires inference. Myers, McDaniel, & Wolber (2000) surveyed a range of PBE systems and identified a common problem, which is that most available tools depend on fixed rule-based systems and users are not allowed to change or update the rules. They point out that Peridot system, for example, that defines layout constraints using a PBE technique depends in its inferences on a set of rules. If extra rules are required or different (customised) examples required to be implemented for more adaptation of the system to the user, a programmer is required to manipulate the Lisp code in which the rules are written. According to them, this is one of the most important challenges in the field of PBE systems, viz., to be able to augment the rules of the inference engine at runtime without the need to manipulate the code directly. According to the literature survey conducted in this research, no existing PBE systems allow the inference rules to be augmented at runtime or without programming.

The notion of learning in PBE has been discussed by some authors (Maulsby & Witten, 1993; Bergman, Castelli, Lau, & Oblinger, 2005 and Castelli et al., 2007). However, their concept of learning is different from the concept described in the paragraph above. For example, Bergman, Castelli, Lau, & Oblinger (2005) and Castelli et al. (2007) use the term ‘learning’ to refer to an inference process that updates the generated documentation (called a ‘procedure’), based on interaction of a user with an Eclipse GUI. The system works based on a “follow-me” technique. When a procedure required to be documented, the user, called “author”, enters the authoring mode and records the interaction with the GUI. DocWizard generates documentation for this interaction (the procedure). Later on, another user uses this documentation by selecting the playback mode and following the documentation step by step. Until this point DocWizard works like macros. However, the second user can divert from the original generated procedure, for example to select another directory to save files other than that used by the author (assuming the documentation procedure contains saving file action). In this case, the second user stops following the original procedure and performs some other actions that do not exist in the procedure. The system compares the old and new actions to discover the differences.

Based on the differences, the system makes inferences to update the original procedure so it covers all the conditions presented by the first and the second users. They call this process “incremental update” as each time a user diverts from the existing procedure, the system updates this procedure. The update takes the form of some conditions (if statements) appears in the procedure to cover the different conditions. This update, based on an inference, is considered to be a form of learning.

The MetaMouse system (Maulsby & Witten, 1993) is a system that learns from the repeated actions of the user so it can detect the next required action. The system watches the user while working and generalises from the tasks. When a task is repeated, it detects the similarity between the current task and a previous one. However, sometimes it generalises wrongly which leads to an incorrect inference. In this case the user interferes to modify the behaviour. This is considered as offering a set of positive and negative examples as introduced in the MetaMouse system documentation in the previous section. MetaMouse depends on the positive examples to learn and generalise and depends on the negative examples to refine the generated and generalised code. MetaMouse works only with the repeated behaviour, which is criticised by Fisher, Busse, & Wolber (1992) as a limitation of the system. As will be seen in Chapter 7, this dissertation documents a tool that learns from the user but is not limited to repeated behaviours or examples. Table 2-2 presents the different domains and contexts in which PBE has been applied according to the review conducted in this research.

Table 2-2: Programming By Example different domains of application

Domain of Application	Example	Author(s)
Diagram editor GUI specification	ISI	(Goldman & Balzer, 1999)
Layout constraints specification between GUI objects	Peridot	(Myers, 1993)
Layout constraints specification between objects other than GUI	MetaMouse	(Maulsby & Witten, 1993)
Context-aware applications	aCAPpella	(Dey, Hamid, Beckmann, Li, & Hsu, 2004)
GUI behaviour applications	Grizzly Bear system	(Frank, 1995)
Games specification	Kidsim	(Smith, Cypher, & Spohrer, 1994)
Event-based Simulation	Gamut	(McDaniel & Myers, 1999)
Authoring Documentation	DocWizard	(Prabaker, Bergman, & Castelli, 2006)
Robotic Applications	N/A	(Argall, Chernova, Veloso, & Browning, 2009)
Spreadsheet Templates	Gencel	(Abraham & Erwing, 2006)
Web Sites	Highlight	(Nichols & Lau, 2008)

The discussion in the previous sections and Table 2-2 show that no applications of the PBE technique in the context of meta-CASE tools have been found

in the literature, apart from one limited application in the domain of GUI visual design editor generation called ISI. The (Goldman & Balzer, 1999) research in the context of meta-tools introduced the term “specification by example” to indicate the specification of the vocabulary and the visual representation of the components (they call this GUI specification) of the required modelling language but not the constraints. This is shown in Figure 2-16 as the “Domain Expert” prepares the constraints that are implemented by the “Analysis Programmers”. This is a completely separated process from the GUI specification, which is according to (Goldman & Balzer, 1999) is achieved using “specification by example”.

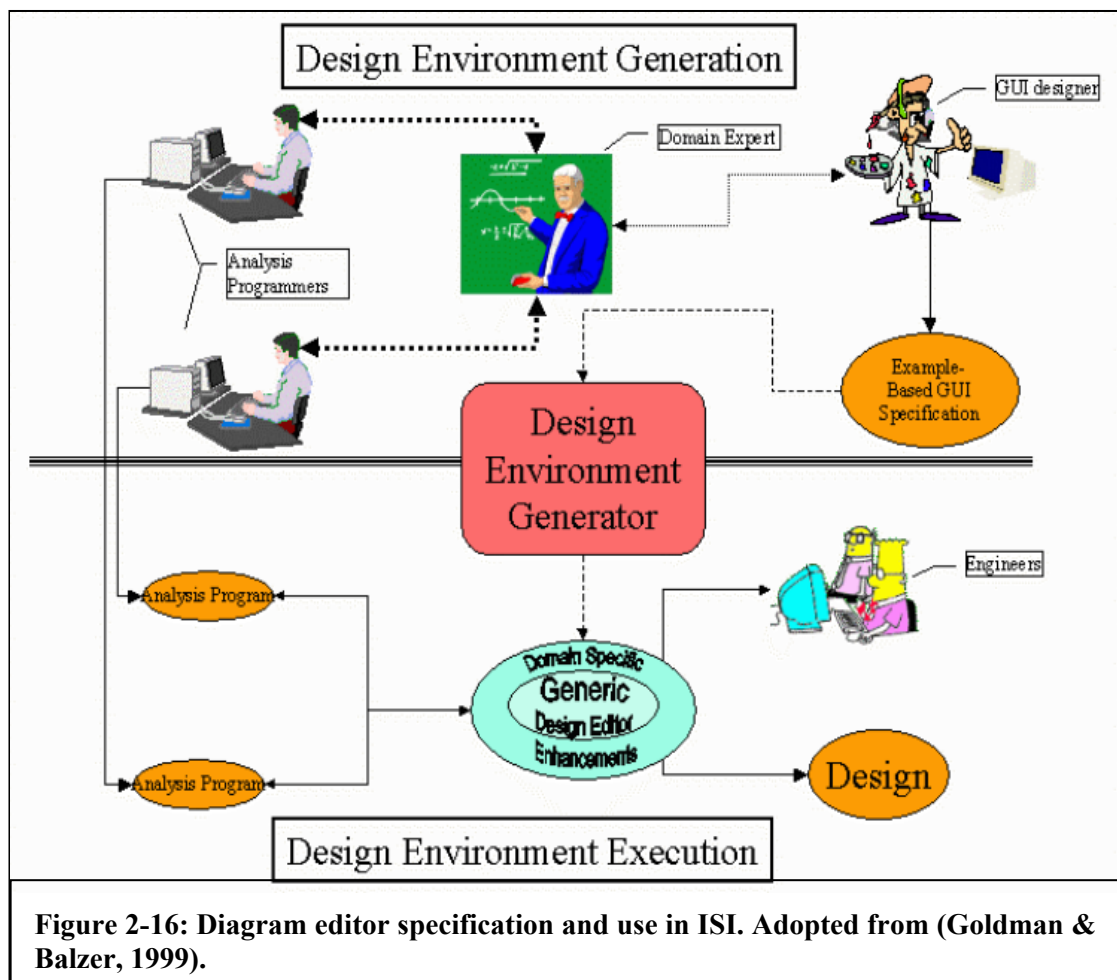
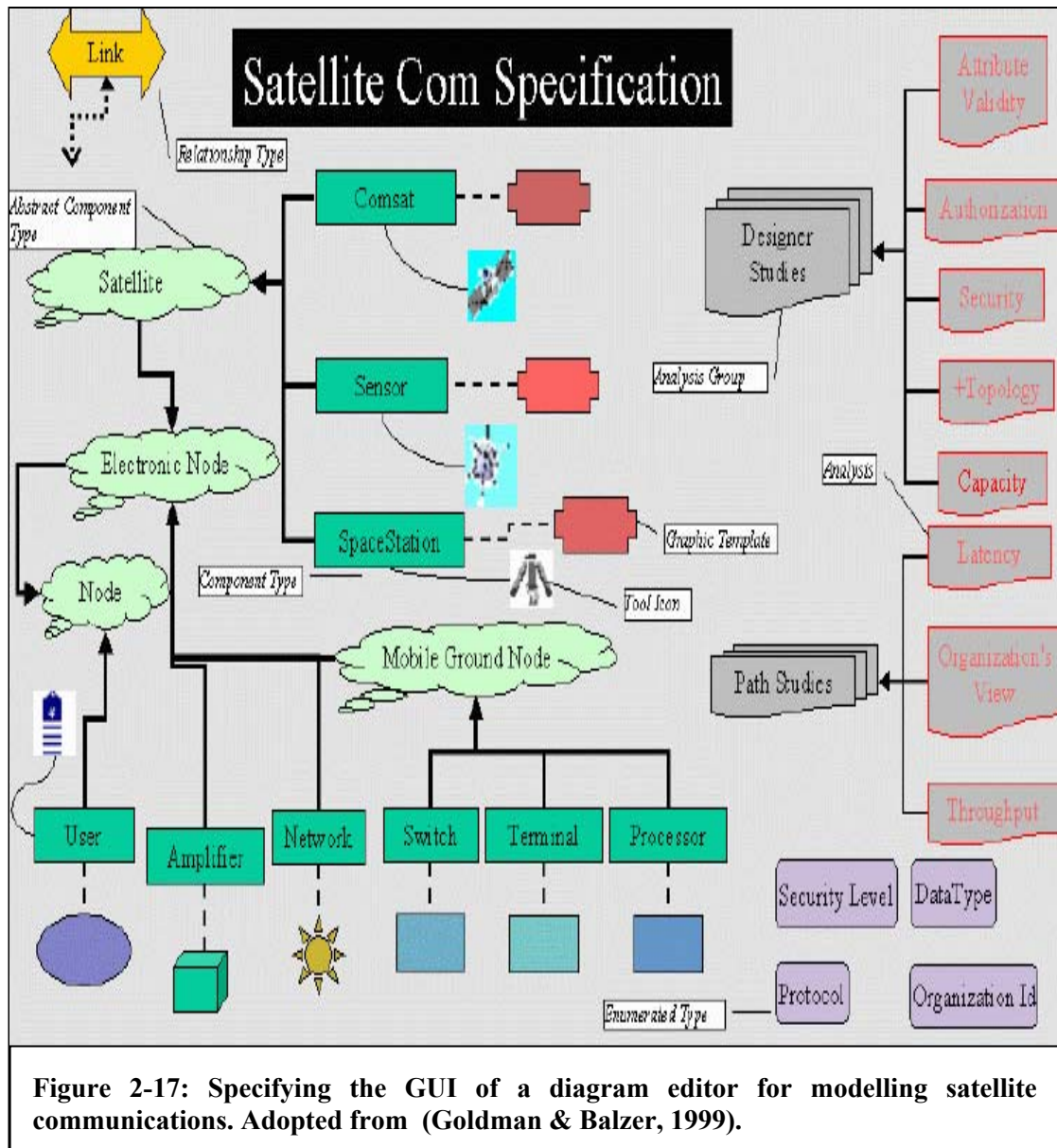


Figure 2-16: Diagram editor specification and use in ISI. Adopted from (Goldman & Balzer, 1999).

The specification is done by introducing a visual meta-model composed of pre-specified components with known meanings. This is considered as specifying the meta-model using a visual language as appear in Figure 2-17. This figure shows specifying the GUI of diagram editor for modelling satellite communications.



They represent the vertex types as rectangles with labels to indicate their concept in the specific domain. The visual representation is specified by connecting these rectangles with “graphic templates”. This is the connection represented with the dashed connection in Figure 2-17. Another connection is used to specify the vertex image in the toolbar, called the “tool icon” which is represented by the curved solid connection in the figure. Edge types are specified exactly the same way. In Figure 2-17 the edge appears at the top left corner and labelled as “link”. In the domain of satellite communications, this is the only edge type available. They call this process “specification-by-example”. However, they specify the constraints (call them analysis) as program code using C++ or Visual Basic. The generated diagram editor

and the developed analysers exchange design information via an object-oriented protocol. The editor and the analysers are decoupled so some commonly used analysers can be predefined to be used off the shelf when required. Figure 2-18 shows the result of the specified diagram editor in Figure 2-17.

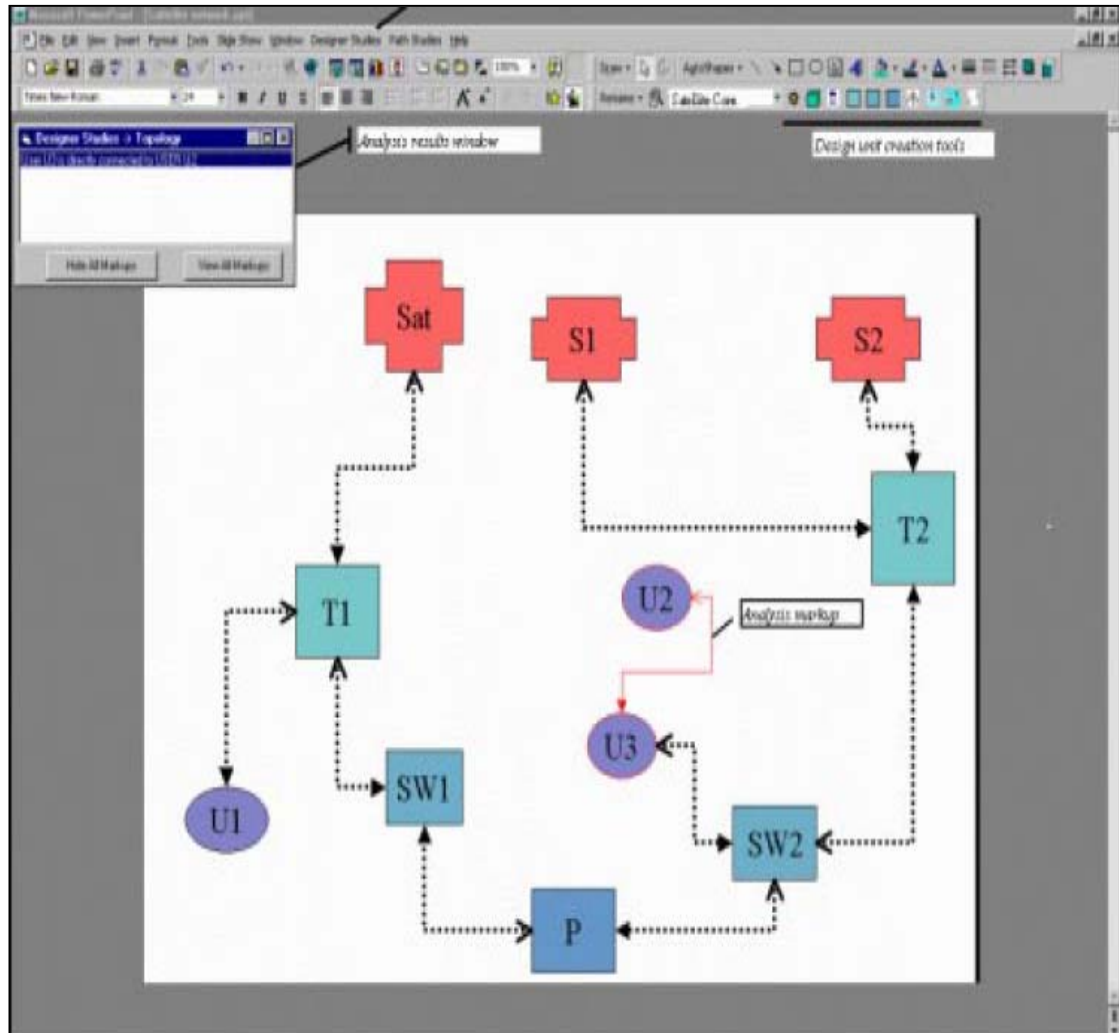


Figure 2-18: The generated specified diagram editor for modelling Satellite Communications. Adopted from (Goldman & Balzer, 1999).

Figure 2-18 shows the visual representation of different components (vertices and edges) and also shows at the top left corner, the result of analysis. This analysis displays in a separate window a list of reports about violations of the specified constraints. The report displays a message that says that “User U3 is directly connected to user U2” which is not allowed. When the designer selects the report, the violation is marked up in the editor and annotated as shown between U2 and U3 in Figure 2-18.

One commercially used application for PBE is macros (used in Microsoft Office, for example). Macros depend on the user to demonstrate a required action or behaviour on a specific application. Based on the behaviour provided, a program is generated that represents exactly the behaviour introduced by the user. Macros, as Myers, (1986) classified them, are considered PBE systems but they are originally under another category, programming *with* example, as there is no generalisation, inference or any artificial intelligent algorithms involved in the process of generating the programs. Macros represent a solution for one of the problems that PBE is also intended to solve, viz., reducing the effort of repeated actions. Because there is no inference in macros, the repeated action example is recorded as it is and repeated as demonstrated exactly without any generalisation. However, many applications that allow the use of macros provide the ability for code editing to generalise the behaviour manually which is considered an efficient method by Myers, McDaniel, & Wolber (2000). Although Myers (1986) has introduced this category of systems (that just memorise with no inference, such as macros) in the category of PBE, this research excludes them. This is because the term PBE is used here to refer to the process of generating a program through a process that includes inference.

2.11 Conclusion

This chapter reviewed literature and presented the background of the research. It started by reviewing the importance of CASE tools (diagram editors in this research) in facilitating software engineering work, their domain specificity and customisability limitations. The solution of using domain specific languages with their associated tools solves the problem according to some authors. However, such tools are expensive to be build, and consequently, they are not typically commercially available. Literature that addressed this problem was reviewed and the solution of using meta-CASE tools, which generate the required diagram editors, was explained. The problem of the complexity of constraint specification in meta-CASE tools was presented. This research addresses this problem by adapting a PBE technique. PBE and its related features, example polarity and learning, were thus reviewed.

Chapter 3

Diagram Editor

Constraint System

(DECS)

3.1 Introduction

This chapter introduces the Diagram Editor Constraints System (DECS), which is used as an experimental meta-CASE tool for conducting this research. It discusses the structure and the features of DECS in general. Since DECS has been built before the start of this research, the chapter presents its initial state and describes the major enhancements introduced to it throughout this research. This chapter also demonstrates how DECS works as a meta-CASE tool system, with some technical details, and shows its meta-modelling process by which a user can define the vocabulary of a target language using the available wizards and part of the syntax and semantics using constraints. Additionally, the chapter introduces the XML-based constraint language that DECS depends on for target language specification and details its distinctive features.

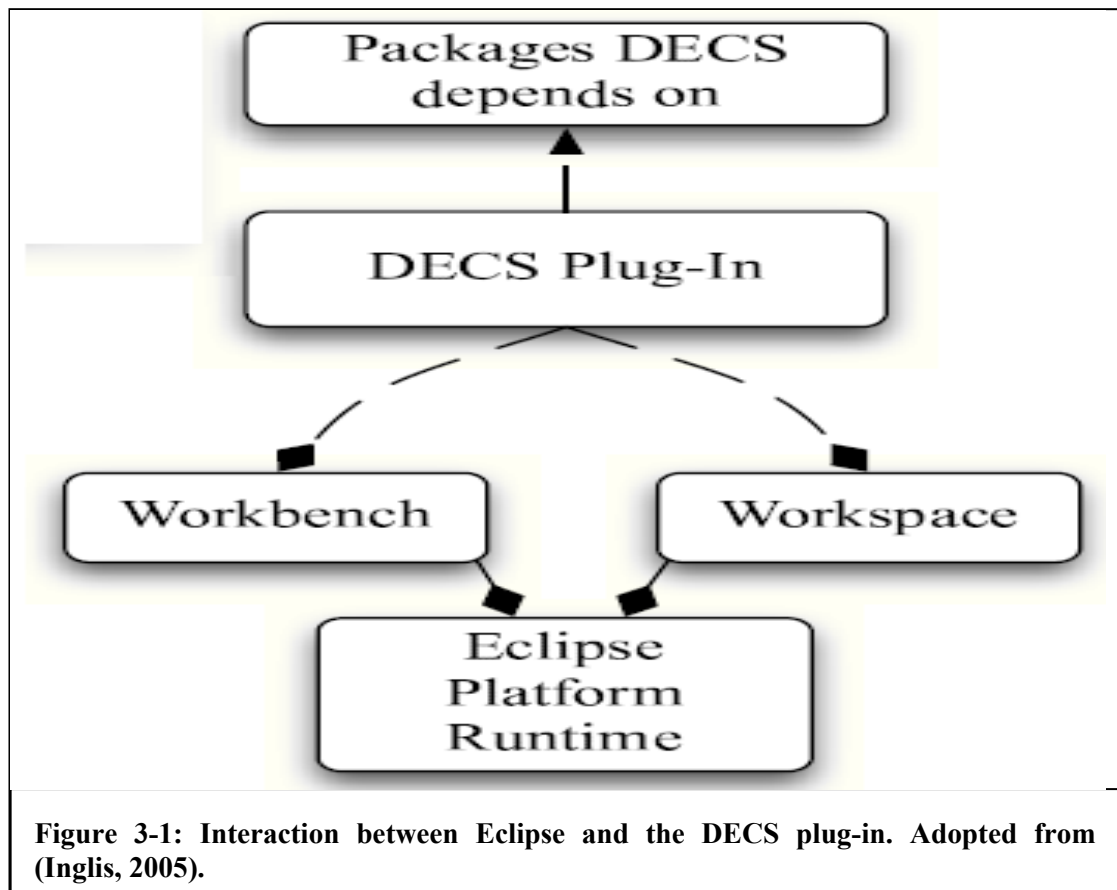
To avoid direct editing of constraint in XML, DECS offers two constraint definition techniques, form-filling and Constraint Specification by Example (CSBE). The first, form-filling, is implemented in DECS in the form of a wizard and tabbed forms. The technique is discussed in detail here with explanatory examples and screenshots. The second technique, CSBE, is described in the next chapter.

3.2 Diagram Editor Constraints System (DECS) Structure and History

3.2.1 Overview

DECS is an Eclipse plug-in meta-CASE experimental prototype initially developed at the University of Glasgow prior to the start of this research. It is a meta-CASE tool that has all the required features to generate a diagram editor. Each plug-in in Eclipse consists of XML manifests that describe the contents of the plug-in to the Eclipse runtime system. The XML manifest file lists the internal and external libraries and dependencies that the current plug-in requires. The XML file also specifies the extension points which specify the new feature that the new plug-in is adding to Eclipse. Any Eclipse plug-in is an instance of Open Services Gateway initiative (OSGi) component, which is part of the OSGi framework. In DECS case,

the extension point is “org.eclipse.ui.editors” which specifies that the new feature is an editor. DECS depends in its work on another framework which is the Graphical Editing Framework (GEF). This framework is composed of two plug-ins. The first is the GEF itself which helps in implementing the graph and manipulate its components. The second plug-in is “draw2d” which helps in rendering and layout the graphics on screen. GEF provides DECS with many features such as editing parts which including connecting vertices and manipulating the edges, creating the figures on the screen, and maintaining and enforcing the editing policies which handles the interaction with the diagrams. Figure 3-1 shows the interaction between Eclipse and DECS plug-in.



This means that in this research, the DECS plug-in was modified and a fork or a branch of code has been added. This, of course, included adding code to the already existing codebase and updating the existing one if required. Figure 3-2 shows the general component structure of DECS while Figure 3-3 shows the use of DECS by different user types. This structure shows that DECS initially is composed of 4 main components. They interact with each other to achieve the meta-CASE tool work.

The meta-modelling component is responsible for managing the specification of the vocabulary, syntax and semantics of the required language. This component interacts with *the repository component* as it saves all the specified vertices, edges and graph types in the repository. A meta-modelling component also reads from the repository component in case the user needs to use previously defined vertices or edges.

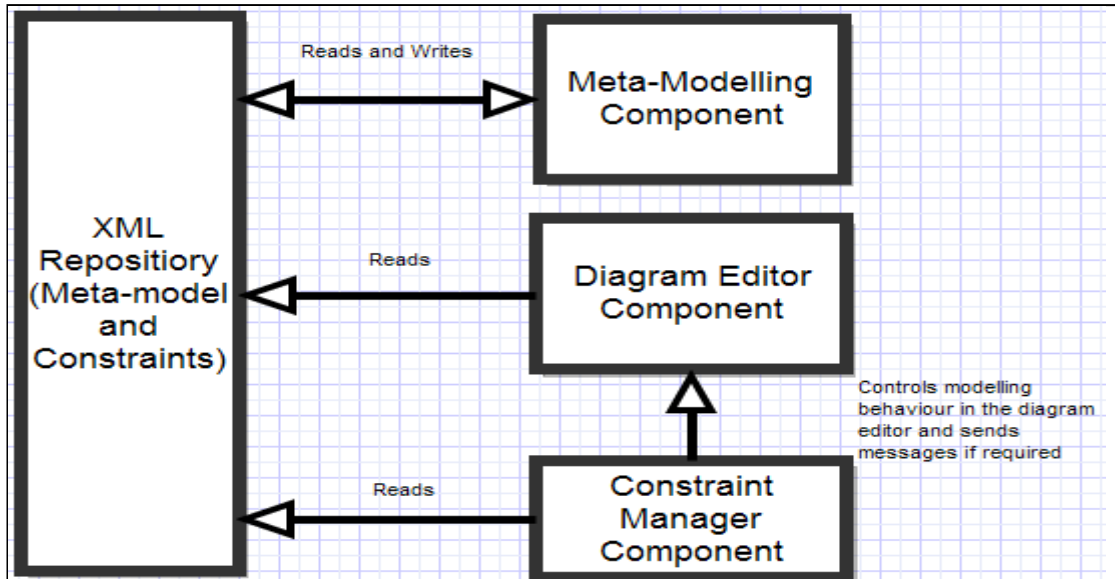


Figure 3-2: DECS Initial Structure and Components Interaction.

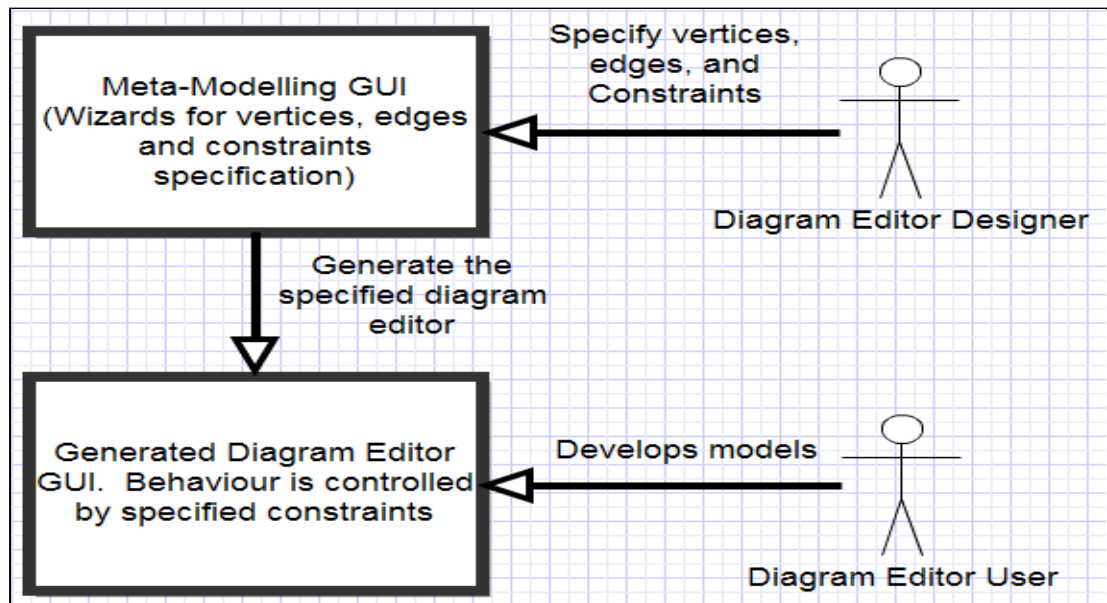


Figure 3-3: DECS Use by different types of users.

The diagram editor component is the one that provides the user with the required editor GUI in which the user can model diagrams of the specified modelling language. Of course, this component reads from the repository component to be able to retrieve the elements of the required graph and their properties. The final component is *the constraint manager* which is the one that monitors the diagrams (models) managed by the diagram editor component and gives decisions to keep the model in a state that does not violate any of the constraints. The constraint manager reads all the required constraints specified in the meta-modelling component from the repository. Figure 3-3 shows the interactions of different user types with DECS. The diagram editor designer works in the meta-level to specify the required modelling language by specifying the participating vertices and edges. Then they specify the constraints using the implemented form-filling technique. The diagram editor is the user that models diagrams using the specified modelling language by the diagram editor designer. It interacts mainly with the GUI provided by the diagram editor component.

3.2.2 Development History

The DECS project originally set out to create a meta-CASE tool that can specify and generate a modelling language editor using a meta-model that depends on drop down menus and constraints instead of graphical meta-models. The project initially started as two undergraduate student projects (McCallum, 2000; Hamilton, 2000) that resulted in a Java-based meta-CASE tool. Additional enhancements to DECS represented with the ability to replay building the model facility and constraint viewer to present the constraints were conducted by Bogie, McCallum, Hamilton, and McGroarty, in 2000 (see DECS website, <http://www.dcs.gla.ac.uk/decs/>). General enhancement has also been introduced to DECS in 2001 by Kristiansen (2001) and McClelland in 2002. A meta-CASE tool similar to DECS project has been made by Nikitas (2005) using C#. The first appearance of DECS as an Eclipse plug-in was of the result of another student project (Inglis, 2005). Integrating the project in the Eclipse IDE using Java allows DECS to benefit from other available plug-ins to Eclipse such as the Graphical Editing Framework (GEF). As Inglis (2005) comments, such an approach provides a suitable base for developing a project for modelling.

Based on that it was decided to adopt his version of DECS to be enhanced (described in Section 3.2.3) and used throughout this research.

3.2.3 DECS Limitations and Enhancements in this Research

The DECS version produced by Inglis (2005) was only able to create editors with a set of basic shapes that include the square, circle, and triangle. Jia (2007) identified some further problems found in DECS:

- the inability to change the size of a shape in the editing area,
- the inability to add labels to edges,
- the inability to edit shapes properties at runtime, and
- the need for more real diagram types instead of modelling just the basic shapes diagrams.

The above problems have been solved during a mini-project at the beginning of this research. The first three problems were defects in the existing code. It was important to solve the last problem because there was a requirement for suitable and realistic software design models to perform experiments on instead of the basic shapes. To be able to solve this problem there was a need to study the way of defining shapes. DECS defines different elements, vertex, edge and graph, using XML files that exist in a repository. Vertex and edge files contain all the required properties for the visual components (or elements, i.e. vertex and edge types) such as the visual representation, background and foreground colours, labels and their colours, and the arrow head positions in the case of edges. The graph file defines the diagram itself and contains all the participant vertices and edges. The graph file is parsed first to generate the diagram editor with its elements. DECS generates diagram editors that are extended from a GEF plug-in (Qattous, 2009).

The vertex and edge specifications are read from file and are parsed at runtime to model the visual representation of the element with all of its specified features. The properties of any element can be updated at runtime without affecting the actual XML files. A Java factory class exists for each of the components and is responsible for building its visual representation when required. This provides an extension point (using inheritance) for any further model components and diagram types in the future.

For the purposes of this research and to provide real modelling language to conduct the experiments on, State Transition Diagram (STD) and Use Case Diagram were developed in DECS. This was a suitable solution for the last problem stated above. Choosing these two diagram types is justified in Section 6.1.2.3.

One main problem that DECS, as an Eclipse plug-in version, suffered from (and not noted in (Jia, 2007)) was the inability of DECS to specify constraints. Accordingly, the generated diagram editors were not able to enforce the constraints that are supposed to be enforced. Since DECS, as a plug-in, has been developed as a student project, the limited time available did not give the chance to develop or implement a constraint language. Accordingly, a suitable XML-based constraint language and constraint checker were developed as a part of this research to overcome this problem and to be able to conduct the required experiments.

The following sections detail the process of specifying a CASE tool (an STD diagram editor) using DECS through its meta-modelling process. The constraint language, and the form-filling constraint specification technique implemented to specify constraints using this language, are also discussed.

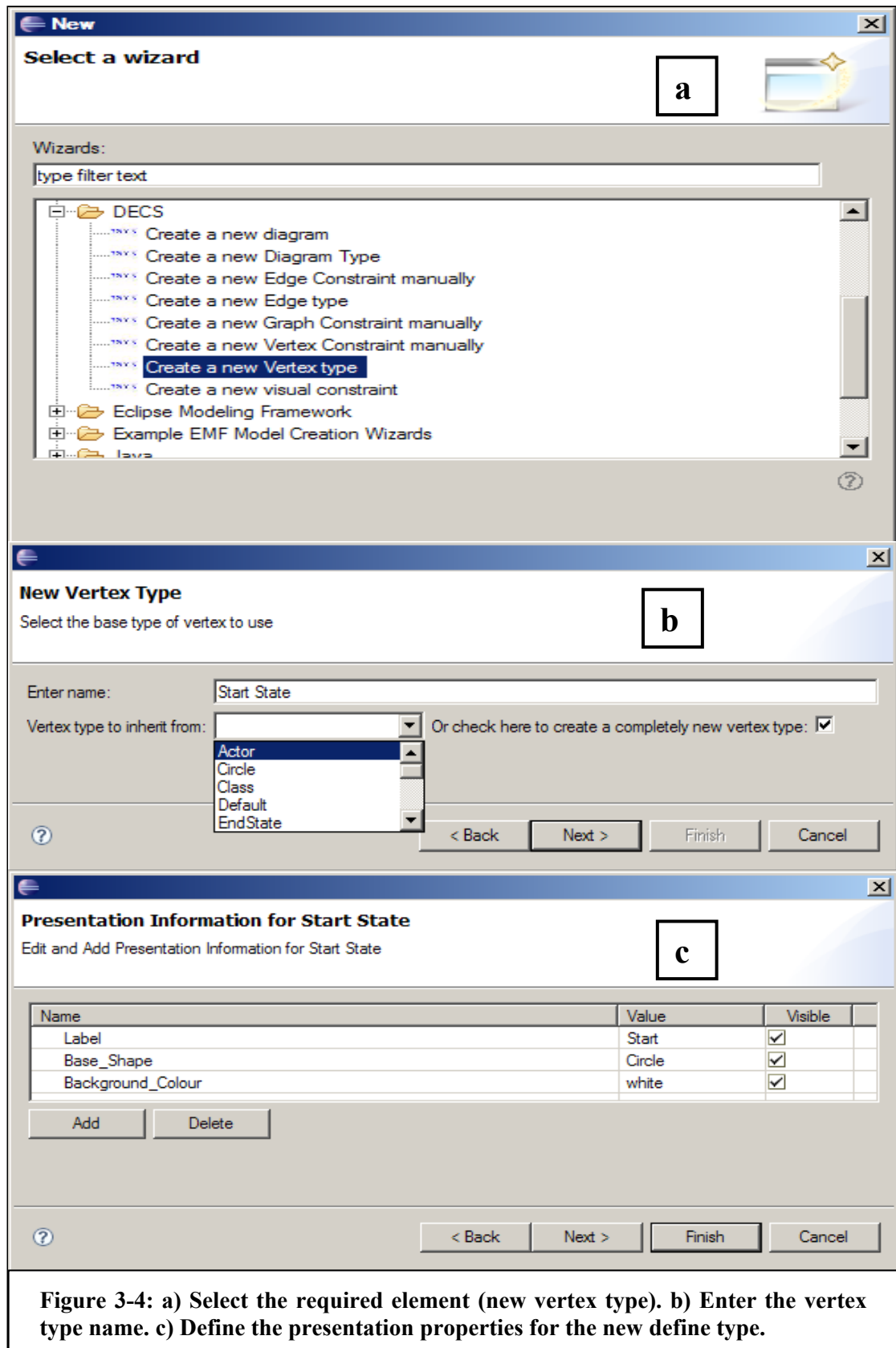
3.3 The Meta-Modelling in DECS

Like other meta-CASE tools, DECS depends on a meta-modelling process for target language specification. The generated meta-model is represented as XML files. In DECS the “editor designer” (Figure 3-3) does the operation of meta-modelling. At the end of this process DECS generates a diagram editor for the specified language. The meta-modelling process in DECS is basic that does not depend on graphical meta-models. It is composed of two parts, vocabulary definition and constraint definition for syntax and semantic specification.

The language vocabulary is defined by selecting the required element types, vertices and edges, which will be part of the target language. These will be the set of available vertices and edges that appear as the modelling language symbols and notations. All the language elements in DECS are defined using XML files and stored in the XML repository (Figure 3-2). This repository is used for communication between different levels of DECS (meta-level and modelling language level) because

the files that are defined and stored in the meta-level will be read and used in the generated modelling language level.

To clarify the vocabulary definition process, the following example shows the process of defining “State Transition Diagram” that has the three vertex types, Start State, End State, and Non Terminal State, and only one edge type, Transition. The “editor designer” defines the required vertices and edges using a wizard. Figure 3-4 shows screenshots for steps of the wizard required to define the “Start State” vertex type. The designer selects the required element to be defined (Figure 3-4-a), enters the vertex name, “Start State”, and specifies if the new vertex type inherits the properties from a previously defined vertex type (in this case it does not) (Figure 3-4-b), and finally manipulates the presentation properties of the defined element by adding, deleting, or changing the values of these properties. This ends up with a new vertex type, “Start State”, defined. A new edge type is defined following the same process using its own wizard. After the designer finishes defining all the required diagram elements (“Start State”, “End State”, “Non Terminal State” and “Transition”), they define the diagram itself. This is done through a wizard that shows the designer all the available vertices (Figure 3-5) and edges. The designer selects the required elements of the diagram to be specified which finishes the process. This generates an XML file in the repository that describes the diagram type.



Select vertices to be used in State Transition Diagram
 Add Vertices for State Transition Diagram

Vertex Name	Use
Actor	<input type="checkbox"/>
Circle	<input type="checkbox"/>
Class	<input type="checkbox"/>
Default	<input type="checkbox"/>
EndState	<input checked="" type="checkbox"/>
myVertexType	<input type="checkbox"/>
NonTerminalState	<input checked="" type="checkbox"/>
Rectangle	<input type="checkbox"/>
SimpleClass	<input type="checkbox"/>
StartState	<input checked="" type="checkbox"/>
Triangle	<input type="checkbox"/>

Figure 3-5: Select the vertices participating as elements at State Transition Diagram.

The “editor user” (Figure 3-3) can at this stage use the specified diagram type by selecting from the list to create a new diagram (Figure 3-6-a) and selects that the required diagram is of type “State Transition Diagram” (Figure 3-6-b). Figure 3-7 shows the generated diagram editor at work.

However, the above diagram type definition is not completed yet. The generated editor with its specified language draws instances of “State Transition Diagram” but without specified syntax and semantics. To complete the meta-modelling process, the “editor designer” should define the diagram syntax and semantics. All the features related to the syntax and semantics of the graph must be added as explicit constraints.

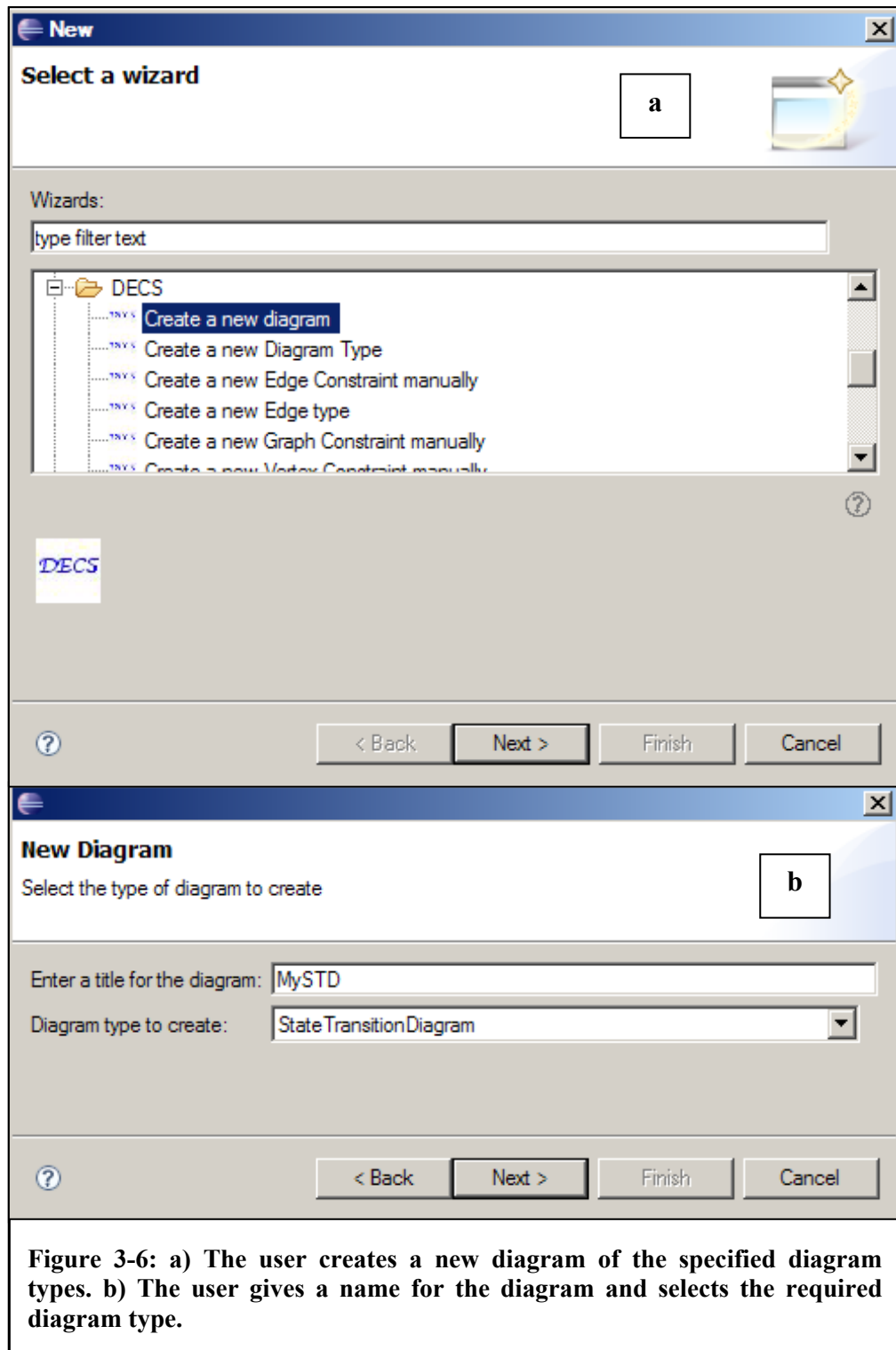
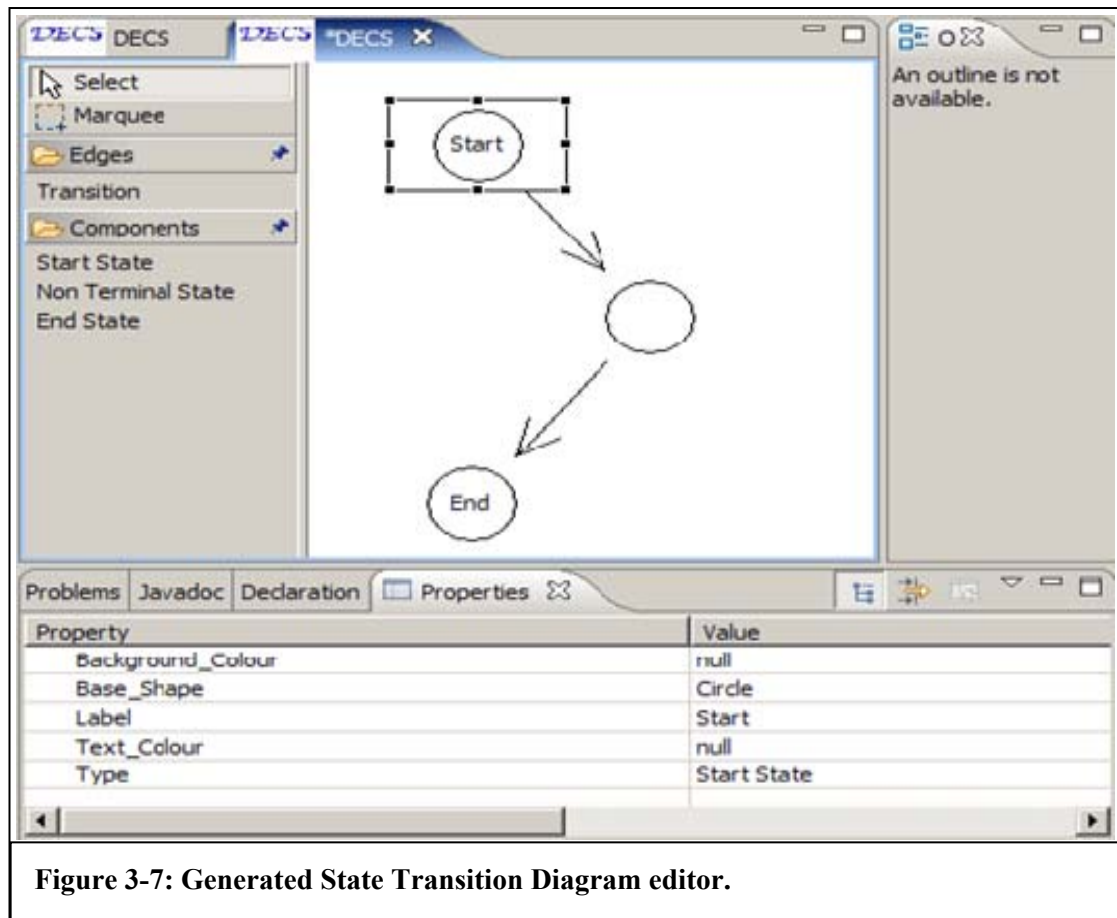


Figure 3-6: a) The user creates a new diagram of the specified diagram types. b) The user gives a name for the diagram and selects the required diagram type.



3.4 The Constraint Language in DECS

DECS depends on an XML-based constraint language which uses property-value assignment to define constraints. The same method has been used by Scott, Horvath, & Day (2000) for constraint definition. The constraint language is used in DECS to specify constraints that form part of the specification of the target modelling language. Figure 3-9 shows one of the constraints that the language is capable of specifying which is *“it is not allowed to connect a vertex of type End State as a source with a vertex of type Start State as a target using an edge of type Transition”*. As the figure shows, this constraint is implemented using different XML constraints. This example will be discussed in detail later in this chapter.

The property-value assignment technique for constraint definition allowed several constraint classifications to be used because classifications can be embedded within a constraint in form of values assigned to properties. In this manner,

constraints in DECS can be considered as adapting multidimensional classification because it currently adopts the following two constraint classifications:

- In DECS, the constraint language classifies the constraints depending on the *scope of the constraint* into vertex, edge, and graph constraints. This classification classifies the constraints based on the starting scope of the constraints into vertex, edge and graph constraints. A vertex constraint is defined as one that starts with the vertex element (the meaning of ‘starting with’ will be discussed later). The edge constraint is similar but starts with edge instead of vertex. The graph constraint specifies constraints that have as their scope the entire graph. Examples of this classification will be discussed later. This classification was adopted because it is simple (i.e., direct and related to the different components available in the context), suits the nature of the system as an XML-based system and solves the problems of other classification as introduced in the above constraint example. According to this classification, the DECS constraint language has three different constraint types that are implemented as three different XML file formats each of which is a format for one of the constraint types. The template structures of these XML files are available in Appendix A. This classification is similar to that introduced in (De Lara & Vangheluwe, 2002). In their meta-CASE tool, AToM³, constraints are divided into two types, “local” and “global”. Local constraints are used to define constraints applied to a specific entity of the graph such as the ‘External Entity’ in an ER diagram. An example of such a constraint is “*the connection of two External Entities by means of a DataFlow is not allowed*”. An example of a global constraint is “*all DataFlow names must be unique*”. Such a constraint is a global one because its effect extends over an entire diagram or a graph. This classification is similar to the one adopted in DECS but with less detail as they combined the vertices and edges constraints into one classification called ‘local’.
- The constraint language also adopts a classification depending on the enforcement of the constraint as “hard” or “soft”. The classification has been also introduced before in (Gray & Welland, 1999). Hard constraints are those constraints that must not be violated at any time during the process of modelling. Soft constraints are those constraints that should be satisfied at the end of a modelling process but not within the process itself. This means that they can be violated to an extent

during the process but the final product model should not be in a state that violates any of them. Such a classification is also introduced in (Vessey, Jarvenpaa, & Tractinsky, 1992) and (Jankowski, 1997). The Argo critic system by Robbins., Hilbert, & Redmiles (1997) adopts and implements both hard and soft types of constraint. A similar behavioural classification of the constraints, based on the priority of the constraints as high and low, is introduced in (Ledeczi, Maroti, & Volgyesi, 2001). For every OCL expression constraint in GME a specific priority is specified. This priority determines the action that should be taken by the constraint manager when the constraint is violated. When a high priority constraint is violated the transaction result of the violation is aborted and an error message is presented, while the system only presents a warning message when a low priority constraint is violated. This constraint classification has been adopted in DECS because it affects the behaviour of the system depending on the constraint enforcement type, which is a required feature. If the constraint is hard, the system will enforce it and prevents its violation which is the desired behaviour for all the constraints. As an example, the constraint *“it is not allowed to have more than one Start State in the diagram”* is more appropriate to be specified as a hard constraint. This is because when the designer adds the second start state vertex, the system should delete it (Figure 3-12). However, in some situations the designer must be given time to provide the correct design that does not violate the constraint; this requires a soft constraint. An example of this constraint type is *“at least one Start State and one End State must exist in a State Transition Diagram”*. This is a soft constraint because the user will add one of the vertices, either start state or end state, then the user will add the second vertex. If the constraint is defined as a hard constraint and the user tries to add a start state vertex at the beginning, the system will delete it and shows an error message saying that both vertices must be in the diagram. The same behaviour will be if the user chooses to add the end state before the start state. This means the user should add both vertices at the same time which is not possible. Because of that, a soft constraint gives the user the required time to modify the diagram in a way that satisfies the constraints; however, the system shows warning messages to remind the user about the violated constraint. This is shown in Figure 3-8.

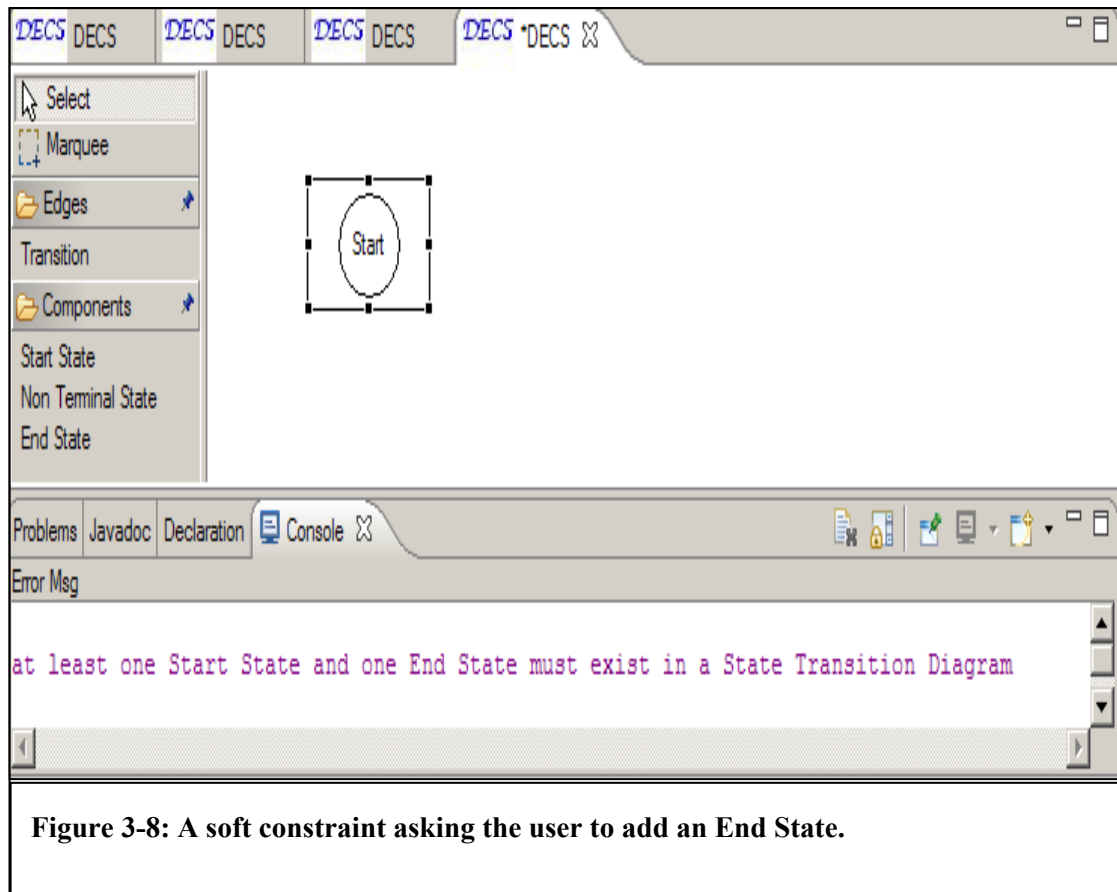
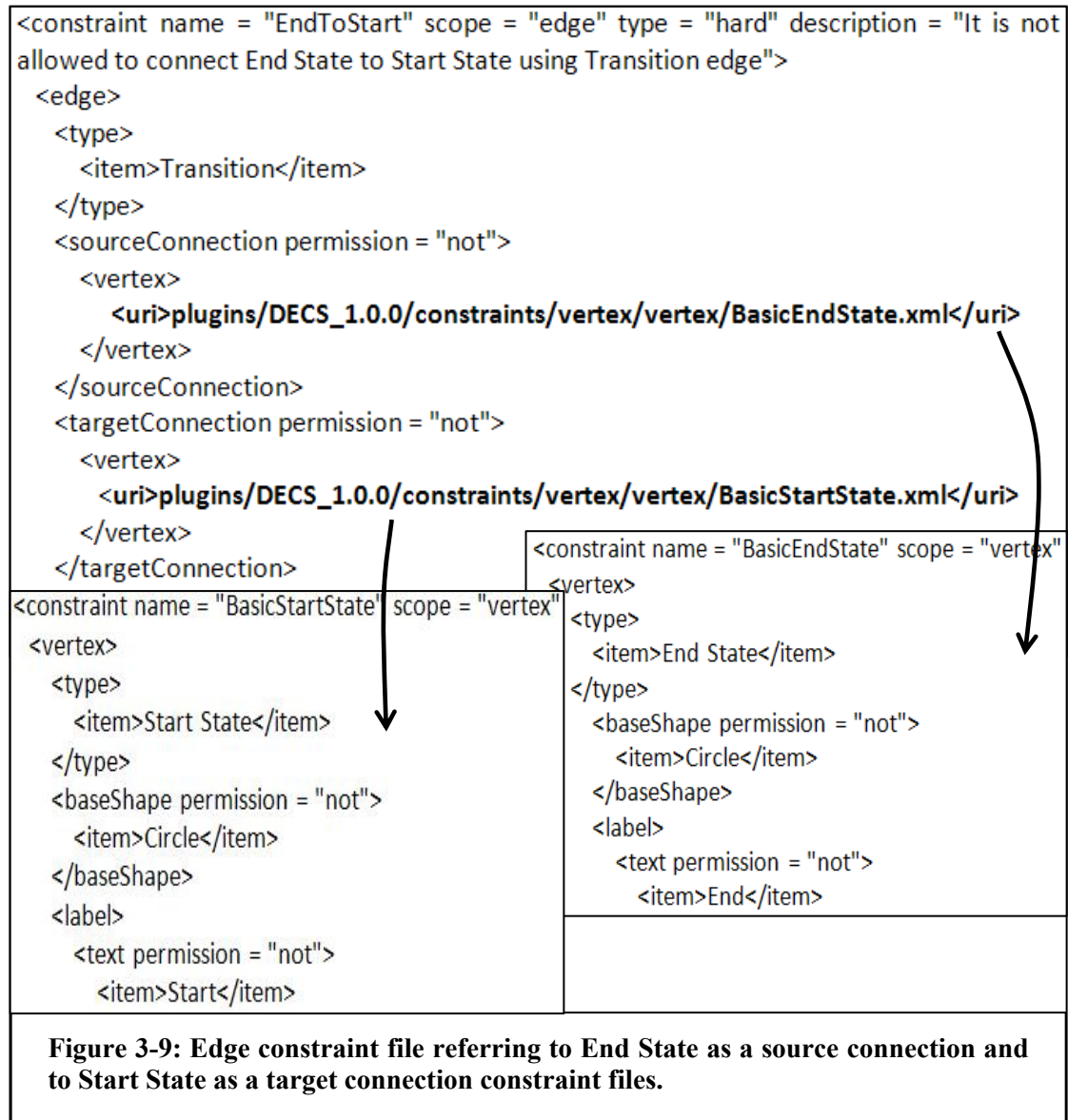


Figure 3-8: A soft constraint asking the user to add an End State.

The property-value assignment feature also provides the ability to refer from one constraint to another. As shown in Figure 3-9, some of the properties in the XML constraint definition take as values URI descriptors which work as references to other XML constraints. This provides the DECS constraint language with flexibility. This flexibility comes from the ability to construct the required constraint using different XML constraint definitions by referencing one to another (Figure 3-9). The referencing feature allows the user to build complex constraint structures by plugging small constraints together. A similar feature has been used in a visual language for event handling definition by Liu, Hosking, & Grundy (2007a).

Referencing also allows the same constraint to be built in different ways depending on the starting point of building the constraint. As an example, in state transition diagram type the constraint, “*it is not allowed to connect a vertex of type End State as a source with a vertex of type Start State as a target using an edge of type Transition*” can be defined either as an edge constraint which starts the definition of the constraint from the edge part then defines the source and target parts or can be defined as a vertex constraint which starts the definition from one of the vertices. In

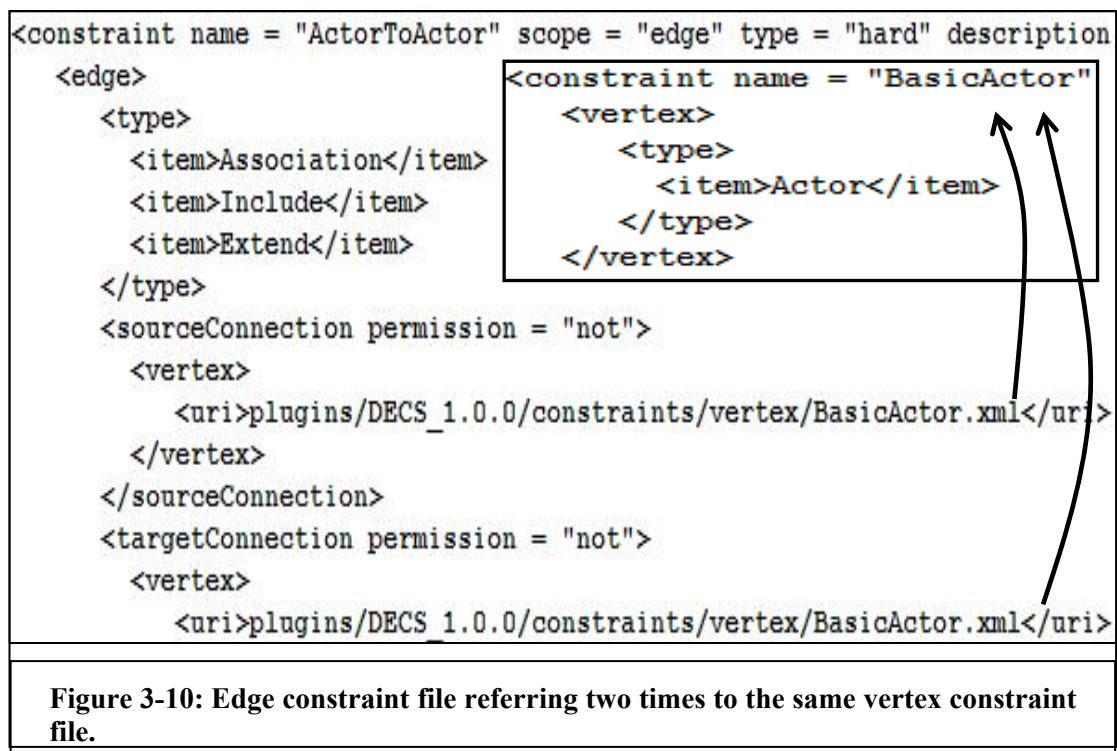
case of the first choice, edge constraint, the user would define an edge that its source is an End State and its target is a Start State. This constraint definition appears in Figure 3-9. Note that in this case the user defines what must not be allowed.



In the case of the second choice, the user has two options; in the first, the user would define a vertex constraint that specifies an End State; its source connection is an edge constraint that specifies an edge of type Transition. The edge target connection property refers to another vertex constraint that specifies a vertex of type Start State. The second option is that the user can specify the constraint in a reverse way by starting the constraint by defining a vertex constraint that specifies a vertex of

Start State, its target connection property is an edge constraint (Transition) that its source connection property refers to a vertex constrain (End State).

The referencing ability also allows reusing previously defined constraints which reduces the effort and time required to define the constraint if more than one element (a vertex as an example) of the constraint has the same description. In this case, there will be only one XML file to describe one of these components and whenever required, there will be a reference for the same file. This also opens the chance to use previously defined constraint components by referring to them instead of defining new constraints. As an example, in the use case diagram the constraint “*it is not allowed to connect a vertex of type Actor with a vertex of type Actor using an edge of type Association, Extend, or Include*” the source and the target vertex types are the same, viz., actor vertex, and the same constraint file will be used. In this case, the values of the source connection and the target connection for the edge constraint could be the same file that defines a vertex of type actor. This means that the user needs to define only one vertex constraint file for both parts of the constraint (the source and the target). This is shown in Figure 3-10. A similar feature has been implemented in GME meta-CASE tool through defining constraints in forms of functions that can be called and reused from within other constraints (Ledeczi, Maroti, & Volgyesi, 2001).



The referencing feature provides one more advantage which is the simplicity of constraint construction and consequently performing a constraint definition task. Based on the divide and conquer concept, it is likely that it is easier to tackle the problem of defining a constraint as a set of small problems than as one big problem. The language also allows complex constraints to be composed using the logical relations AND and OR between URI constraint references which helps in constructing complicated constraint structures (that involve several vertices and edges with logical relations between them). This feature also helps in defining more than one constraint at the same time. As an example, in the constraint defined in Figure 3-10, it is possible, using the OR logical operation, to add another URI for the target connection property that refers to a vertex constraint of type Use Case. In this case the constraint will be *“it is not allowed to connect a vertex of type Actor OR a vertex of type Use Case with a vertex of type Actor using an edge of type Association, Extend, or Include”*. In addition, the referencing feature helps in simplifying the constraint generalisation feature which will be discussed in detail in chapter 7. Because of all the features described above, this constraint specification language was developed and implemented in DECS and used in this research.

3.5 Constraint Specification

All the defined constraints are stored in a repository. At runtime (when the “editor user” uses the generated editor), the ***constraint manager*** (Figure 3-3) pulls all the defined constraints from the store and converts them into Java objects. This is done with the help of a recursive descent parser implemented to be able to follow the references in each constraint. The Java object for each constraint is constructed using the wrapper design pattern and maintained within the constraint manager component which helps in constraint evaluation later on. The constraint manager monitors the work of the designer (editor user) in the generated diagram editor and validates every action against the maintained constraint objects. Every time the “editor user” modifies the diagram, the constraint manager scans the constraints to assert that the updated state of the diagram does not violate any of the available constraints.

If a violation of a constraint is triggered, the constraint manager behaves depending on the violated constraint classification (hard or soft). This constraint

classification specifies the behaviour of the constraint manager. If the constraint is a hard constraint, the constraint manager undoes the last action performed in the editor to return the diagram to a legal state and shows the user a message justifying the action. However, if the constraint is a soft constraint, the constraint manager only shows the user a warning message describing the violation. The adoption of this classification allows the user to specify the required behaviour of the constraint. The user can specify the type of the constraint, if hard or soft, using the attribute “type”. Figure 3-10 shows the definition of “hard” constraint.

The last thing to describe in the constraint specification is the “description” attribute (near the top of the constraint definition following the “type” attribute). This is the message that the constraint manager presents to the user when the constraint is violated which is used to describe and clarify the violated constraint.

3.5.1 Constraint Specification Using the Form-Filling Technique

As a part of meta-modelling process, a DECS user should be able to define constraints using the language described above. To achieve this, it is possible to manually create or edit the constraint. However, to provide a more convenient technique for constraint specification, the form-filling technique, in the form of a wizard and tabbed forms, has been introduced in DECS. Justifications of using this technique have been introduced briefly in chapter 1. The form-filling technique was adopted in this research instead of any other constraint specification techniques introduced in the literature review because it is the only documented typical technique for constraint specification in meta-CASE tools. As shown in the literature review, the other techniques for constraint specification in meta-CASE tools are experimental and research techniques. However, form-filling is considered as a typical technique instead of being a research one and is used in the commercial meta-CASE tool MetaEdit+ (MetaCase, 2009). Additionally the form-filling technique is common in different meta-CASE tools instead of being implemented in just one tool for research purposes.

Using the form-filling technique, the user defines a constraint by filling in the required values for the constraint properties. There is a space for each property of the constraint file in form of a textbox or a checkbox. This means that the wizard is a

GUI reflection of the constraint definition structure. If another constraint is needed to be referenced, another form can be filled as each form represents one constraint. Referencing can be done either by defining a new constraint through calling a new tabbed frame form from within the current one (Figure 3-11) or by browsing the available constraints to reuse a previously defined one. Both definitions lead to referencing files (the constraint URI, as shown in figures (Figure 3-9 and Figure 3-10)).

Figure 3-11 shows the steps to define the constraint introduced above: *“it is not allowed to connect a vertex of type End State as a source with a vertex of type Start State as a target using an edge of type Transition”*. This helps in comparing between constraint specification by direct editing the XML constraint file and using the form-filling technique to specify the constraint.

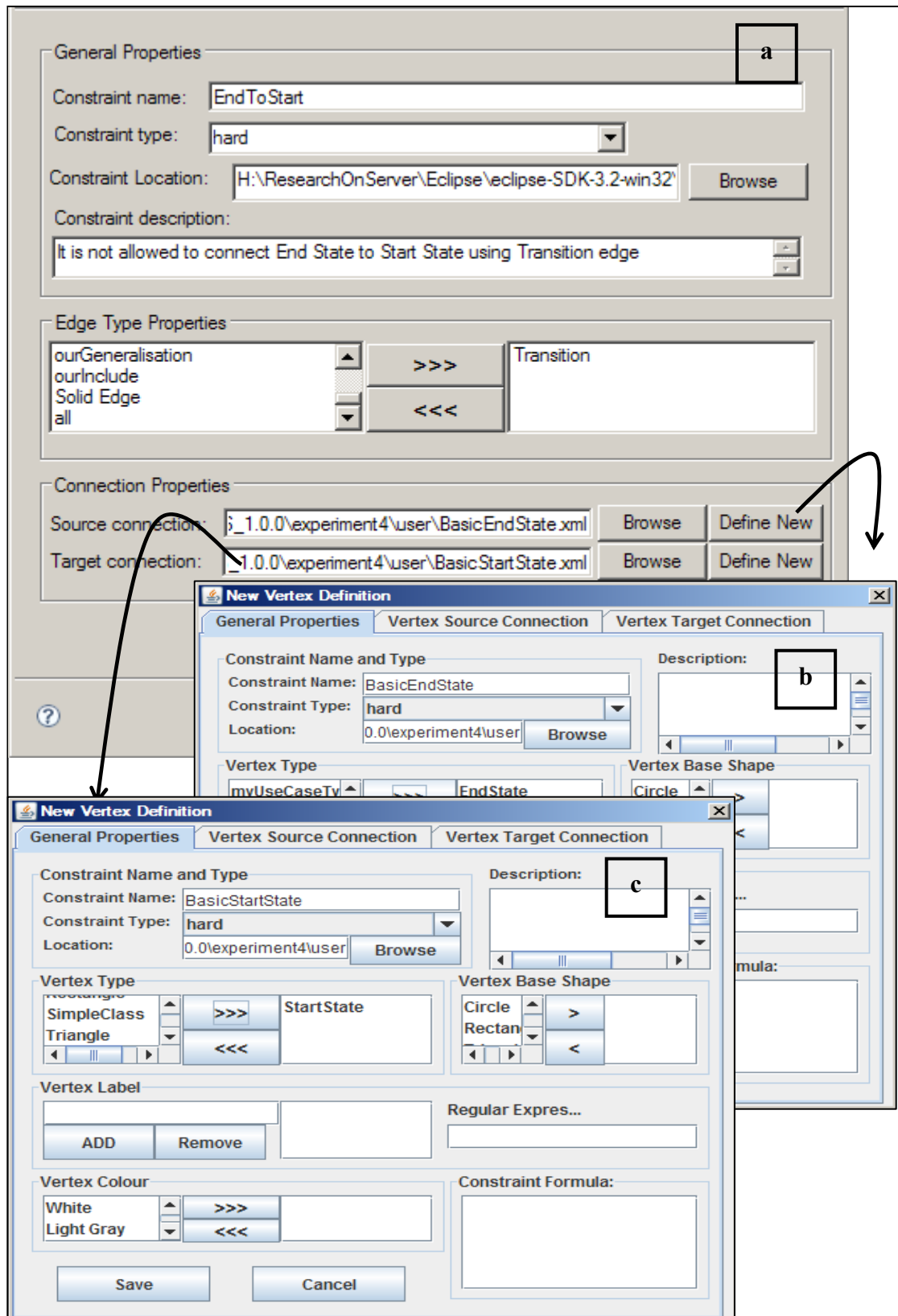
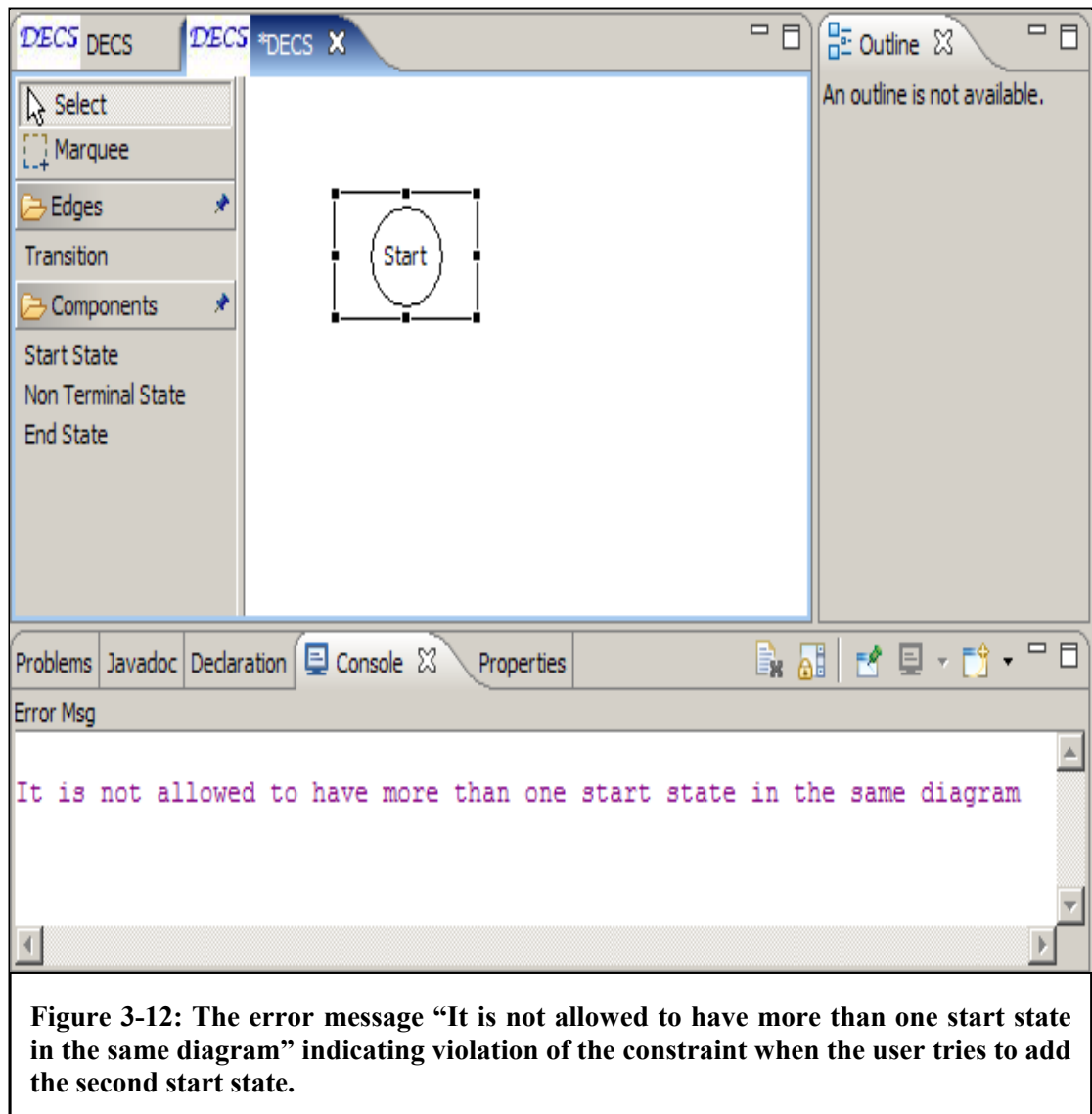


Figure 3-11: a) Transition edge constraint form (wizard) that defines two new constraints as references. b) End State vertex constraint form (tabbed-frame) to define referenced source connection constraint from the edge constraint. c) Start State vertex constraint form (tabbed-frame) to define referenced source connection constraint from the edge constraint.

To complete the picture Figure 3-12 shows a generated diagram editor that models a State Transition Diagram. One of the constraints specified to define this diagram editor is *“it is not allowed to have more than one Start State in the same diagram”*. Figure 3-12 shows that the user already added one start state and has tried to add one more. This violates the constraints and the constraint manager, as shown in Figure 3-10, deletes the second start state and presents an error message to the user, which is the text defined as the description of the constraint.



3.6 Why DECS?

DECS has been selected for use in this research for the following reasons:

- The main reason for choosing DECS for this research is the fact that it is dependent on constraints as the dominant part of its meta-modelling process. This gives the required flexibility for testing the application of the CSBE technique for constraint specification.
- DECS has been implemented partially before the start of this research which helped to save time and effort in meta-CASE tool issues, particularly implementation and code customisation, that is out of the scope of the research.
- DECS has been implemented as an Eclipse plug-in which gives the opportunity to extend it as an experimental meta-CASE tool for research purposes worldwide. Additionally, Java is used for implementation which made it easy for the research to deal with because of the author's strong background in this programming language.
- The dependence of DECS on the GEF plug-in provided a suitable GUI for the generated editors and saved the time and effort of implementing features such as drag and drop and drawing edges.

In addition to all of the above, it is believed that DECS is unique as a meta-CASE tool in its dependence on constraints as the main source of target language specification. All the reviewed meta-CASE tools utilise constraints but not to the level that DECS does. It was noticed that DECS using constraints is able to specify many target languages include those used in this research (State Transition Diagram and Use Case Diagram). Since the research is not about the sufficiency of the constraints to specify the target languages in meta-CASE tools, no studies were conducted in this direction. However, in a trial to evaluate the ability to depend on constraints to define a modelling language, the ZigBee domain specific modelling language was analysed theoretically as a trial to see if it could be specified in DECS using only the constraints.

ZigBee is a network protocol that has some rules that restrict the participant nodes to follow. The following specifies the syntax and semantics of the ZigBee protocol as extracted from (Stevanovic, 2007).

- 1) ZigBee classifies devices into only three types based on functionality,
 - a) ZigBee end-device.

- b) ZigBee router.
 - c) ZigBee coordinator.
- 2) Exactly one coordinator must exist in ZigBee.
 - 3) It is possible to have one or more routers in ZigBee (not compulsory).
 - 4) It is allowed to have a connection between coordinator and end-device.
 - 5) It is allowed to have a connection between coordinator and router.
 - 6) It is allowed to have a connection between router and end-device.
 - 7) It is not allowed to have connection between two end-devices.

From the above conditions, it can be concluded that it is possible to define a domain specific modelling language for the ZigBee protocol using DECS. This will be possible because all the constraints specified above can be represented using the constraint language implemented in DECS. The difference of specifying the above language using DECS and using any other meta-CASE tool is the simplicity of depending only on the constraints in case of DECS. However, this needs extra research to ensure that constraints alone can capture different concepts in other domains.

3.7 Conclusion

The Diagram Editor Constraints System (DECS) is an Eclipse plug-in meta-CASE tool system that depends on an XML repository to store the specification of the required target language. It depends on a wizard for vocabulary definition. For the target language specification, DECS depends on constraints. DECS has a structure with an XML repository responsible for the communication between the meta-level and the diagram editor (target language) level. It also has a constraint manager that is consulted to ensure that the models in the diagram editor do not violate the defined constraints (the syntax and semantics of the specified target language). For constraint specification, DECS depends on a flexible XML-based constraint language.

In DECS, the user can specify constraints using the form-filling technique that is represented as a wizard and tabbed forms. DECS depends mainly on constraints to

specify the required modelling language. Since this research aim is in the domain of constraints, DECS is a suitable tool to be used to conduct this research.

The next chapter discusses a novel technique, Constraint Specification by Example (CSBE), invented as part of this research, with its associated features including its synergistic approach. This will be supported by the presentation of some technical details of the DECS inference engine.

Chapter 4

Constraint

Specification by

Example (CSBE)

4.1 Introduction

In the previous chapter, DECS was introduced as a meta-CASE tool prepared as a prototype to conduct the empirical studies of this research. Its XML-based constraint language was described, along with the DECS constraint manager. Form-filling has been implemented in DECS as one constraint specification technique. To be able to validate the thesis statement and answer the first question stated in the introduction, there is a need to implement the CSBE technique into DECS.

This chapter discusses the CSBE model developed as part of this research and discusses its novel features that adapt the PBE technique to the constraint specification task. These features include the use of an interactive synergistic approach, remodelling and visual generalisation within the process of inference, positive and negative examples and a system learning approach. Taken together, CSBE combines its features into a distinctive variant of PBE not found in any previous systems.

The chapter also introduces the CSBE implementation in DECS. This clarifies the practical application of the model and its features at work, illustrated using examples and screenshots. It also shows some other features associated with the implementation such as constraint visualisation and inference engine transparency features. This discussion is combined with the introduction of the inference engine and the implementation of its rules and their different types.

4.2 Constraint Specification by Example

The ordinary PBE technique for specification or configuration depends on providing one or more examples of the required program to the PBE system. The system then infers the program by generalising the examples (Argall, Chernova, Veloso, & Browning, 2009). In the context of constraints, the user introduces one or more examples that express the required constraint and the system attempts to infer the intended constraint. Note that, since inference is an essential part of the process, macros, and other similar systems, cannot be considered candidates for CSBE.

4.3 Visualisation-Oriented Constraint Specification

DECS and its CSBE technique depend on a distinctive feature, in the context of meta-modelling, of expressing the constraints using the same elements (vertices and edges) of the target language (language to be specified) instead of using a different representation for these elements. It is believed that this feature is a participant in increasing the intuitiveness and the visualisation of the constraint. This is because the user defines the constraint using the same visual representation of the target language in the examples.

Throughout the literature on meta-CASE tools, it can be noticed that the meta-model is specified using some model paradigm, typically one that is different from the target language visual representation. Scott (1997) uses hypernodes to express the model while the KOGGE meta-CASE tool uses an extended version of an ER diagram (EER) (Ebert, Süttenbach, & Uhe, 1997). KOGGE's meta-model specification includes even the visual representation of the target diagram elements (vertices and edges).

Visualisation in the meta-model using the same objects as in the target language has been introduced by Draheim et al. (2010). They call this visualisation feature 'visual reification' and they define it as "the notion that metamodels are visualized the same way as their instances". This includes some visual representation of the target model in the meta-model. They justify its introduction in the meta-modelling language by the requirement for intuitive meta-modelling features. They focused on the principle of intuitiveness as a solution for the problem of unavailability of meta-modelling in the context of business modelling tools. They introduced the idea of visual reification to make meta-modelling more intuitive in order to address the complexity of the meta-modelling process.

4.4 CSBE Model

Some PBE systems depend in their work on very little interaction between the user and the system, leaving all or most of the work to the system. In this case the user will have very limited control over the generated program (Castelli, Oblinger, & Bergman, 2007). Other systems allow interaction between the system and the user at

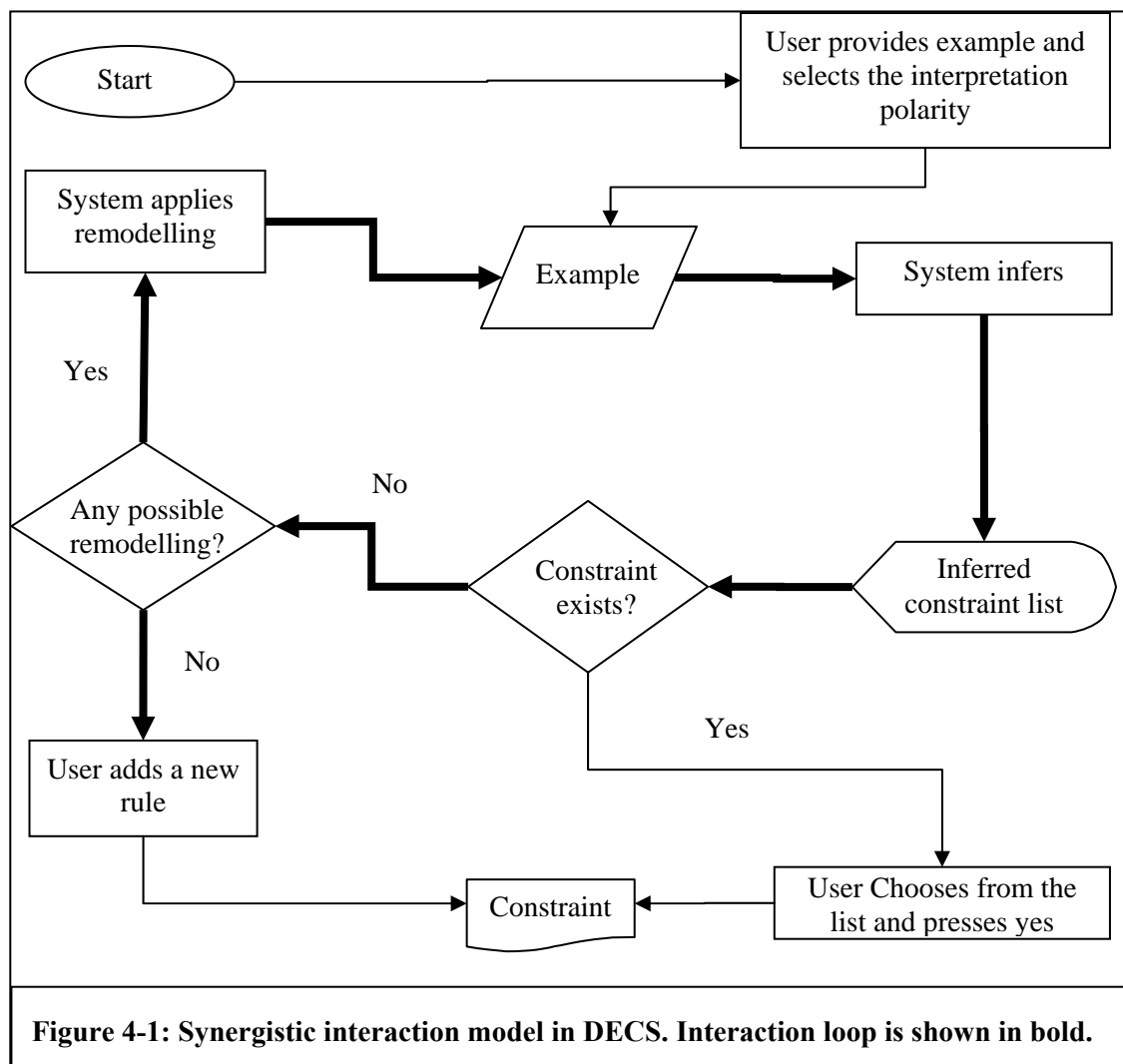
different levels. This interaction can be extensive and complicated (depends on many concepts) as in the case of Gamut (McDaniel & Myers, 1999) that requires the user to define what the system should do and refine the behaviour by specifying what the system should not do using hidden objects, that can be seen only during the design time but not at the runtime,. The interaction can also be complicated in the case of aCAPpella system (Dey, Hamid, Beckmann, Li, & Hsu, 2004) that requires the user to specify the important parts of the behaviour so the system can generalise only from these parts. The interaction can also be very limited as in Peridot (Myers, 1993) that only allows the user to accept or reject one inference of the system at a time.

The CSBE technique in DECS requires user involvement in the inference and generalisation processes. It does not restrict the user's work to confirming or rejecting a single inference; however, it also does not require him/her to provide many complicated examples (that depends on many concepts) with hidden objects to define a constraint such as in Gamut (McDaniel & Myers, 1999). It depends on and adapts a synergistic approach that creates an interaction between the user and the system to specify these constraints.

A collaborative, or synergistic, user-system interaction approach has been used before in the context of PBE by Hudson & Hsi (1993). However, that approach depends on a user-centric heuristic search space by generating new possible solutions based on user choices. They also depend on two rules for generalisation and generating solutions based on combinations between them. In DECS, the synergistic approach depends, in its simplest form, on the system providing all the inferences of the introduced example and on the user helping by selecting the intended constraint. However, in some cases, the synergism between the user and the system extends beyond this. Figure 4-1 shows a model of this approach clarifying different possibilities and interaction alternatives.

The model describes the synergistic interaction approach between the user and the system to specify the required constraint. The first step is that the user introduces an example. The user also selects, explicitly, a polarity for the example. The polarity issue will be discussed in detail later on, but for now CSBE allows the user to specify two types of example polarities, positive or negative (discussed in Section 4.5.1). A

positive example shows what must hold and a negative example shows what must not be the case. After this point, the example is ready to be interpreted by the system.

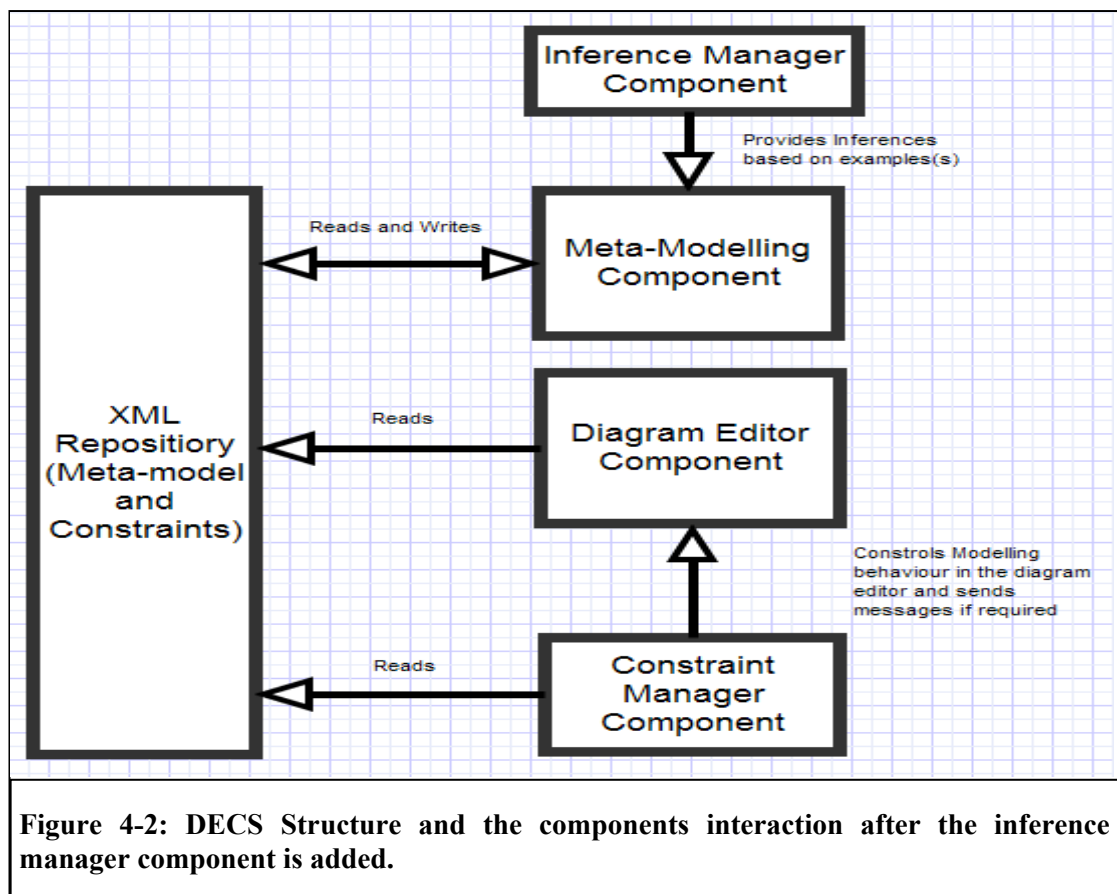


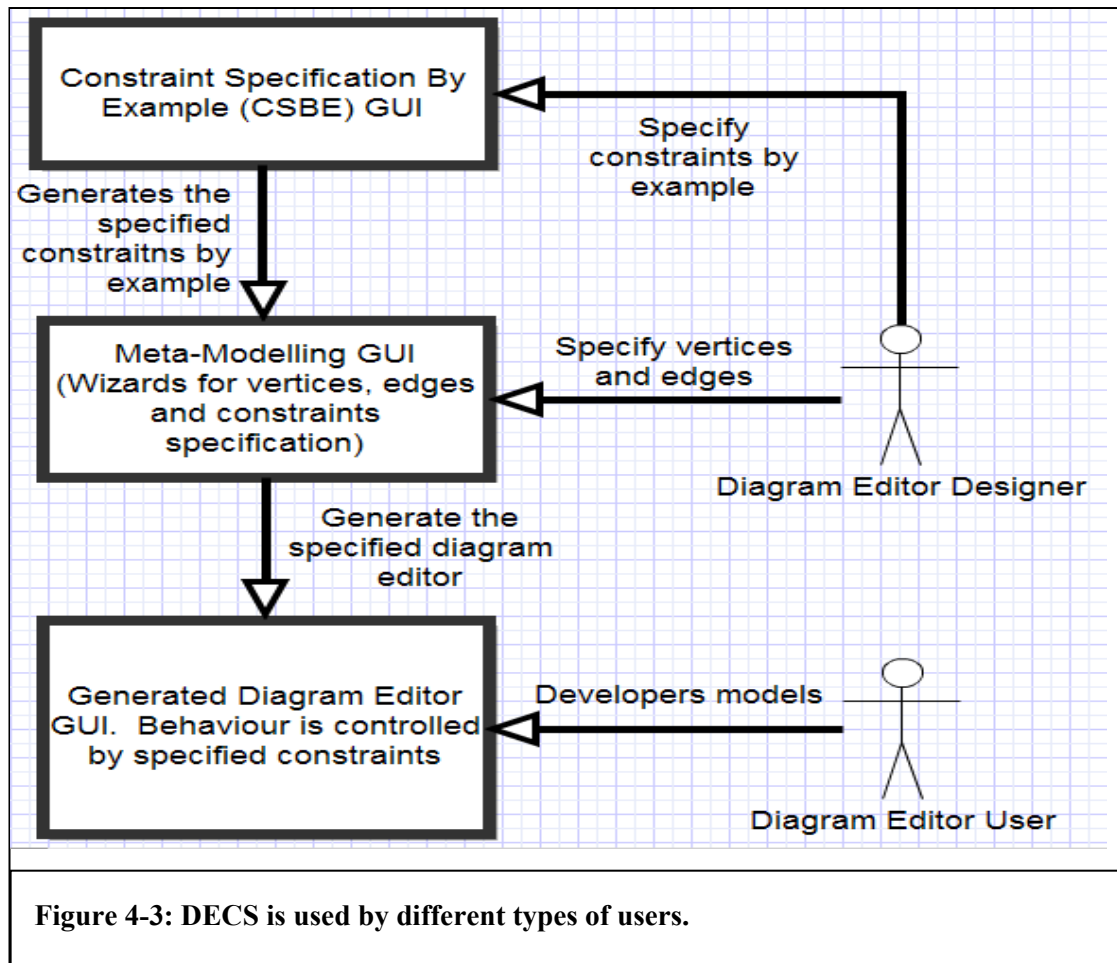
The system takes the example as input and the inference engine works on it to generate an inferred constraint list. This list is presented to the user who searches the list to find the required constraint. If the user finds the constraint, they select it and the constraint is generated. If the constraint does not exist in the inferred constraint list, the system tries to refine the introduced example by remodelling it visually. This allows the system to infer from the modified example (discussed in Section 4.6). If there is no possible remodelling, the user adds a new rule to the inference engine or, in other words, teaches the system how to infer from the example (discussed in Section 4.8). If there is a possible remodelling, then the system performs it, which generates another example. The new example is used as input again to the inference

engine to generate another constraint list. The process is repeated until the required constraint is achieved or the user abandons the attempt. The following sections discuss the implementation of this model in DECS and detail its features.

4.5 CSBE Design and Implementation

Figure 4-2 introduces the DECS structure after CSBE is implemented into it. Note that Figure 4-2 is similar to Figure 3-2 in the previous chapter but with an extra component, Inference Manager, that is added to implement the CSBE model as a constraint specification technique in DECS.





The difference in the components structure of DECS is reflected on the use of DECS by different types of users. Although there is no difference between Figure 3-3 and Figure 4-3 regarding the user types, the diagram editor designer, in addition to the form-filling technique, has an additional alternative for constraint specification in the meta-modelling level which is CSBE. This gives the diagram editor designer the option to interact with either the form-filling technique GUI or the CSBE GUI for constraint specification purposes. However, the diagram editor user is not affected at all by the changes introduced to the meta-level.

The inference manager is the component that maintains the inference engine implemented in DECS. It contains all the inference rules as strings which allows extending them as required through direct scripting. For every string in the rules, there is an associated Java class to perform the work. Although the rules can be extended using a language very close to natural language, there is a limitation of the mapping availability between the script and the Java classes. Any extension to the

rule expression language itself requires an extension of the factory method that is responsible for building the required Java class.

The inference engine implemented in DECS is an adaptation of an open-source one described in (Sazonov, 2004). Although the inference engine is almost completely changed, it was the base that DECS' inference engine depends on. The input of the inference engine is the diagram that is modelling the example (i.e., the example that expresses the constraint). The inference engine extracts features from this diagram (the example) through triggering the rules. The triggered rules are executed which generates the constraint list. More details about the work of the inference engine are in the following parts of this chapter.

The best way to explain and clarify the implementation of the model is through scenarios. Three scenarios are introduced below: one to show the simplest form of interaction, the second to show a more complicated interaction with generalisation (so it is complicated because it involves the generalisation concept), while the third is the most complicated, involving system learning (so it is complicated because it involves the learning concept). A Use Case diagram editor will be used as the target editor in these scenarios. Note that the scenarios describe the process using simple constraints (do not involve many vertices and edges). This is intended to simplify the process and understand it fully. However, this does not mean that the system is unable to define other than these simple constraints. Other complicated constraints (involve many vertices and edges) can be defined because of the flexibility of the constraint language as shown in the previous chapter.

The first scenario for explaining the approach shows the task of defining the constraint:

“It is not allowed to connect two vertices of type Actor using an edge of type Association”.

To define this constraint, the user first introduces an example to express this constraint. For this constraint, the user uses a negative example that shows two vertices of type Actor connected to each other using an edge of type Association. This example is shown in Figure 4-4-a.

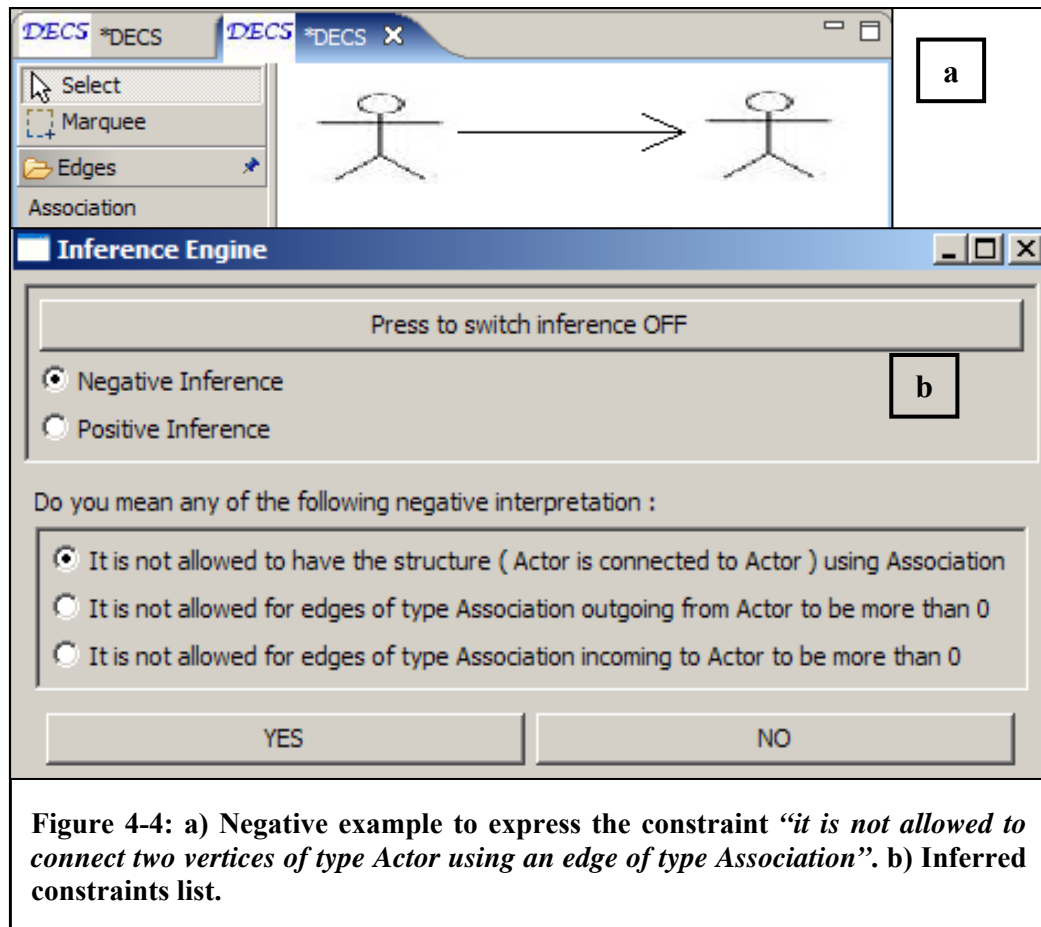
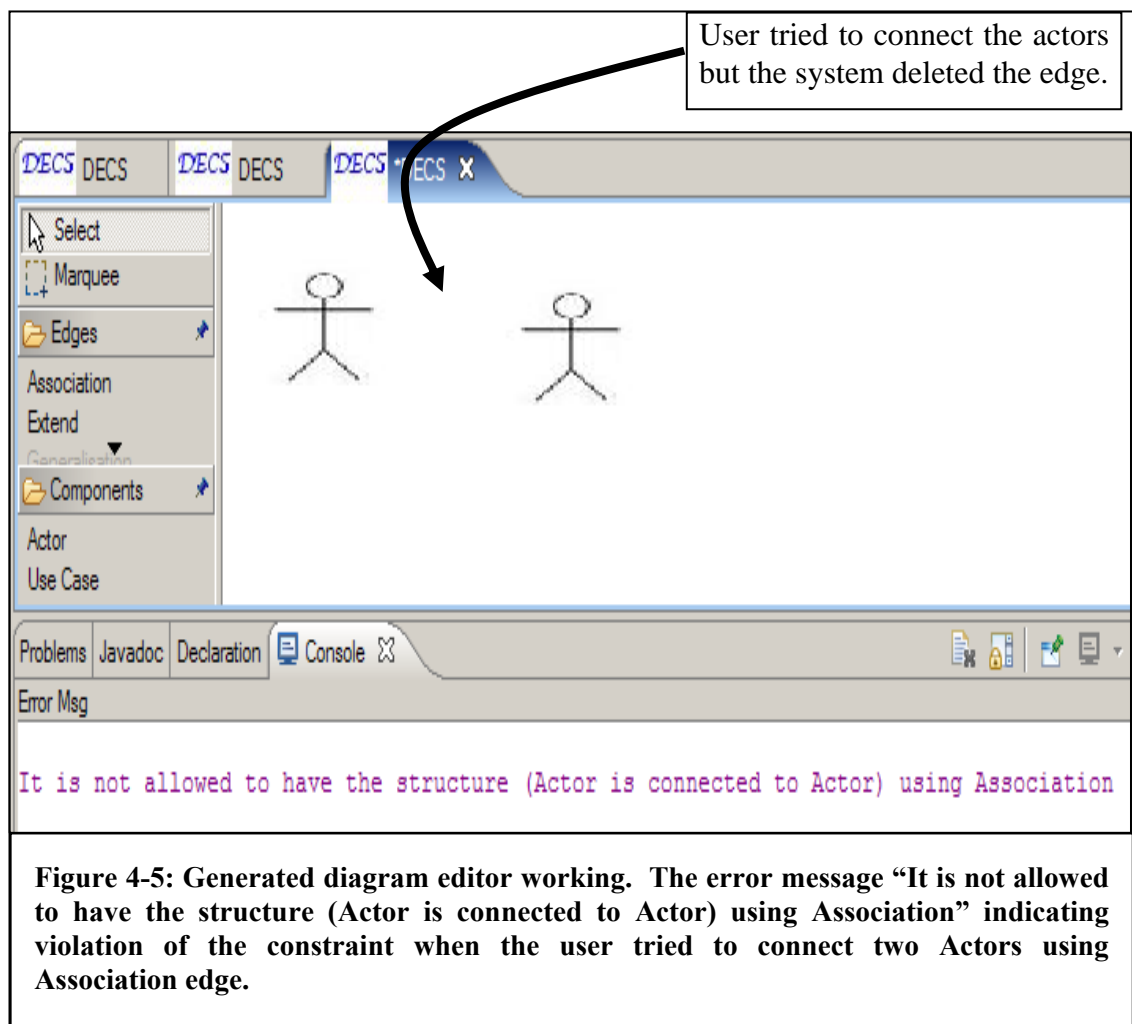


Figure 4-4: a) Negative example to express the constraint “it is not allowed to connect two vertices of type Actor using an edge of type Association”. b) Inferred constraints list.

Once the user introduces the example, s/he asks the system to infer by pressing the button “switch inference ON” (see Figure 4-4⁵). The system attempts to infer the constraint from the example. Inference here can be thought of as an interpretation process, since the system tries to interpret the example. Because the example supports several interpretations, the system infers all the possible interpretations, according to its knowledge, and presents these inferences to the user in form of a list with items attached to radio buttons (Figure 4-4-b) which means that the user can only choose one of the constraints presented in the list. The system asks the user if the list contains the intended interpretation (the required constraint) or not. In Figure 4-4-b the required constraint is shown as the first choice, by coincidence, and selected by

⁵ Note that, in Figure 4-4, the user has already pressed the “ON” button, so the dialogue shows “Press to switch inference OFF”. In general, a user can toggle the inference engine off or on, as desired.

default. The system turn ends here and passes control to the user. The user selects the intended constraint and confirms the selection by pressing “YES”.



The system then shows another dialog box asking the user to give a name for the constraint and to select a URI to store the constraint file in. The system finishes the process by generating a constraint in the DECS constraint language and storing it in a file in the specified location. Just to complete the picture, Figure 4-5 shows the generated diagram editor with the new constraint applied in it. When the user creates a model that contains two Actors and s/he tries to connect them using an Association edge, the system deletes the connection and shows the user a message explaining the constraint.

Before going to the second scenario that shows some novel DECS inference manager features, some features in the first example will be clarified and discussed.

4.5.1 Positive and Negative Examples

A positive example represents the desired state or what must hold, while a negative example represents an undesired state or what is not allowed. In the example above, the user introduced a negative example to show the system what is not allowed (viz., connecting two Actors using an Association edge).

In its initial form, DECS was developed to handle negative examples only, as many typical constraints are naturally expressed in the form “*it is not allowed to ...*”. The example presented above takes this form. However, it was realised that some constraints are easier⁶ and more natural to express positively (using positive examples). The constraint “*it is not allowed to have less than one Actor in the diagram*” (or “*there must be at least one Actor in the diagram*”) (or “*the lower bound number of Actor is 1*”) is an example of such a constraint. To express this constraint positively, the user only needs to provide an example of one Actor. In other words, the example says, this is the desired case. Figure 4-6 shows this constraint with the positive interpretation. Figure 4-7 shows the negative interpretation of the same example.

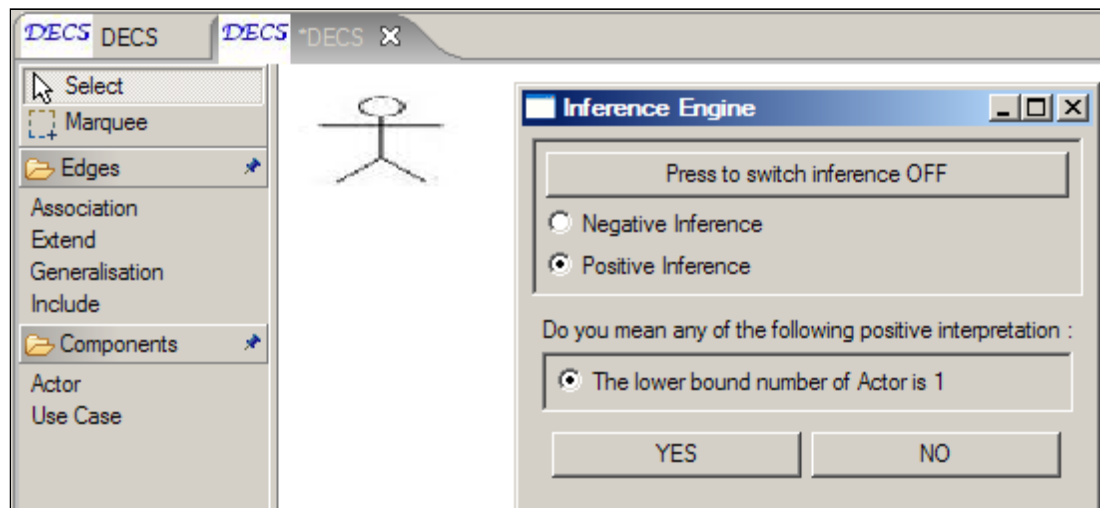


Figure 4-6: Positive interpretation (inference) the introduced example.

⁶ easier = less mental effort

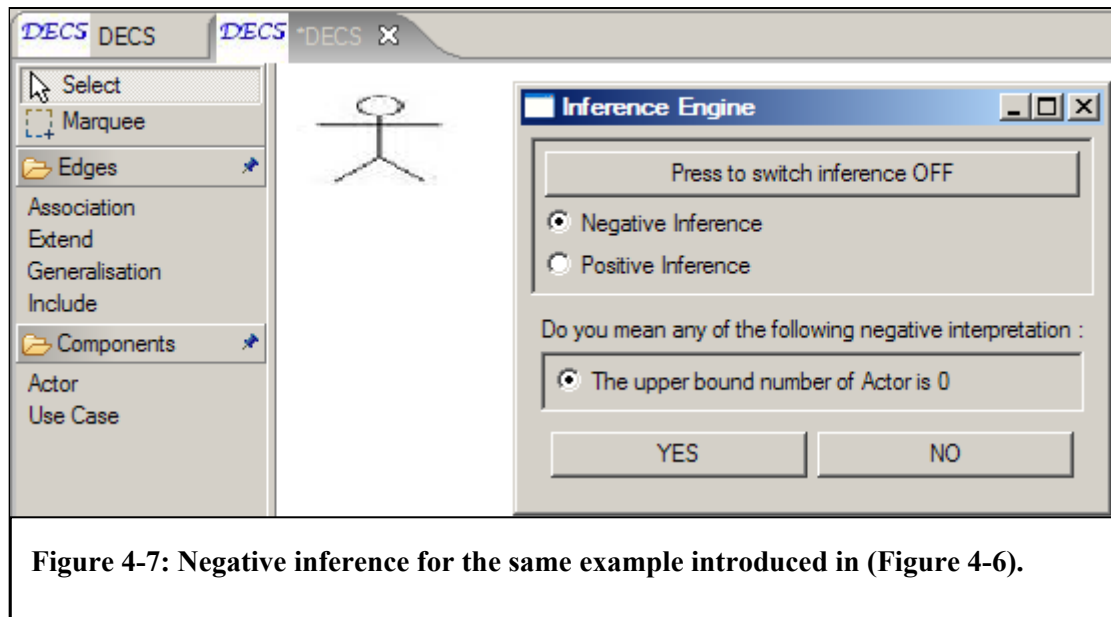


Figure 4-7: Negative inference for the same example introduced in (Figure 4-6).

The negative interpretation of the same example can be explained in the constraint *“It is not allowed to have any Actor in the diagram”* (or *“the upper bound number of Actor is 0”*). Although this constraint is not likely in practice, this is what DECS infers negatively from it.

Many previous PBE systems support both positive and negative examples. Peridot (Myers, 1993) depends mainly on positive examples to define GUI constraints; however, some constraints need to be expressed using negative examples. Some other PBE systems introduced positive and negative examples as one unit that are used together to refine the specification. This is clear in Gamut (McDaniel & Myers, 1999) that uses explicit negative examples to exclude behaviour from a generalised one. MetaMouse (Myers, McDaniel, & Wolber, 2000) uses implicit negative examples to refine the behaviour through conditional branches in the code.

In DECS positive and negative examples are used explicitly with the direction of the user to enforce the interpretation polarity of the example as intended. They are not used to refine the behaviour of each other; instead, each of them is used to define a constraint by itself and separately from the other.

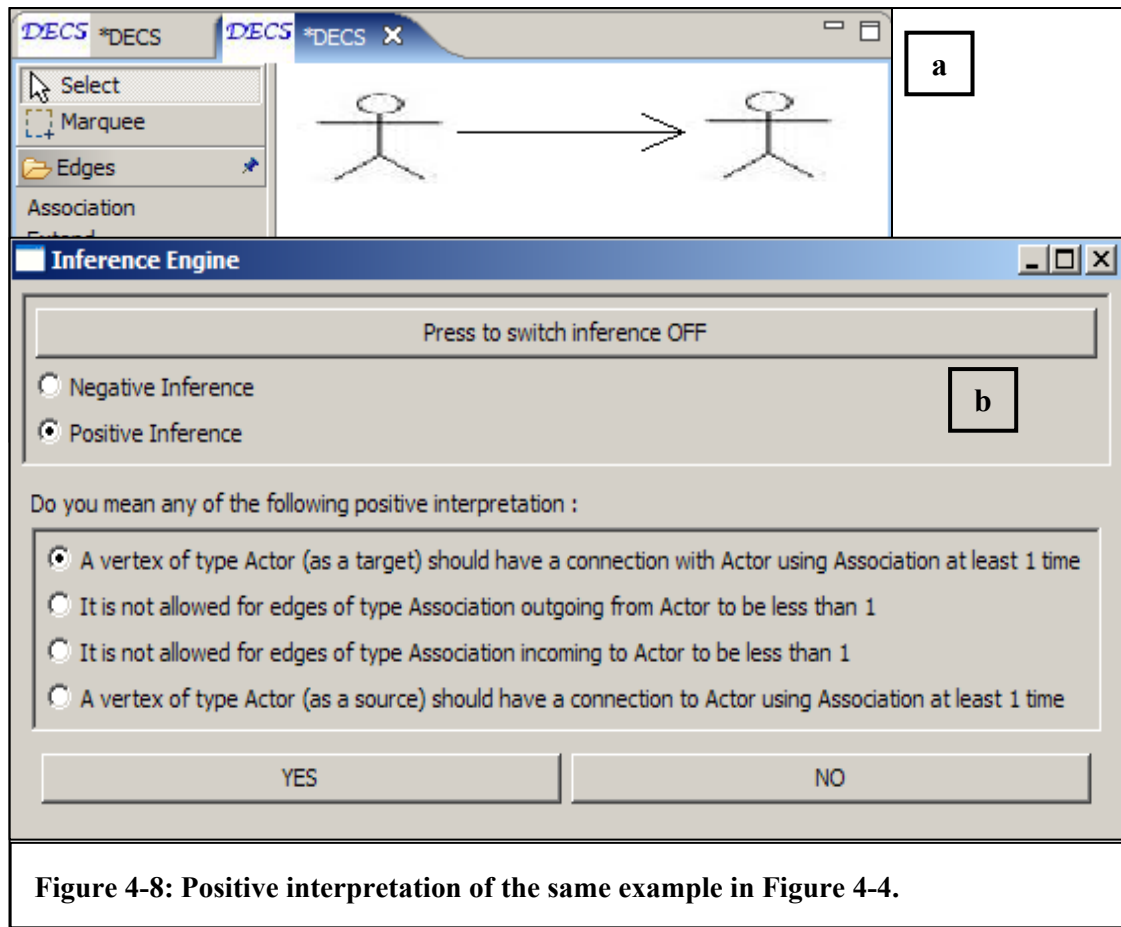


Figure 4-8: Positive interpretation of the same example in Figure 4-4.

Any example in DECS is presented either positively or negatively but not as a combination of both in the same example. This gives the opportunity to express a constraint using an example with the preferred polarity (positive example or negative example). It also reduces the complexity of using both polarities to express the same constraint by combining both types together in the same example to refine the behaviour for each other. Since DECS cannot guess the polarity of an example which can hold different interpretations in either of the two polarities, the user must help the system by choosing the required interpretation polarity. This is done by selecting either a negative or positive interpretation using the radio buttons, as shown in the Figure 4-4-b and Figure 4-8-b. The first figure shows the negative interpretation while the second shows the positive interpretations for the same example (Figure 4-4-

a and Figure 4-8-a)⁷. The same is for Figure 4-6 and Figure 4-7 as they show the positive and negative interpretation for the same example.

4.5.2 Inference Over States and Actions

In the previous examples, DECS inferred the constraints from the state of an example. In other words, the system makes inferences based solely on the state of the example (vertices and edges) at the moment of inference. However, it is possible to combine the example state with an action performed on this state to complete the example. This means that in some cases, the state may be not expressive enough to express the required constraint. As an example, consider the constraint:

“It is not allowed to have less than two Actors in the diagram”.

This constraint can be expressed by introducing an example of two actors. This is a positive example to show that at least two actors must exist in the diagram. The same constraint can be expressed negatively in two steps. First, the user introduces two actors in the diagram, and then s/he deletes one of the actors. The system in this case interprets the example negatively by interpreting that the action taken by the user (deleting the actor) should not be allowed, and infers that at least two actors must be in the diagram. To reach this inference, the system considers both the state of the model prior the action, which is here the two actors, and the action performed on this state, which is the delete action. Using this feature, the user can express a constraint using an example that depends on presenting a state and an action over it. This feature has been implemented in DECS for two reasons. First, it enables a user to express constraints according to his/her preference which should increase DECS' ease of use. Second, it enriches DECS with different ways to express the same constraint which enhances its expressiveness power.

⁷ Note that these are not meaningful diagram states for a Use Case diagram.

Consider the following example that illustrates the use of the state-action feature to enable the use of a single polarity. The constraint (expressed in Figure 4-6) can be expressed negatively instead of positively using the state-action technique.

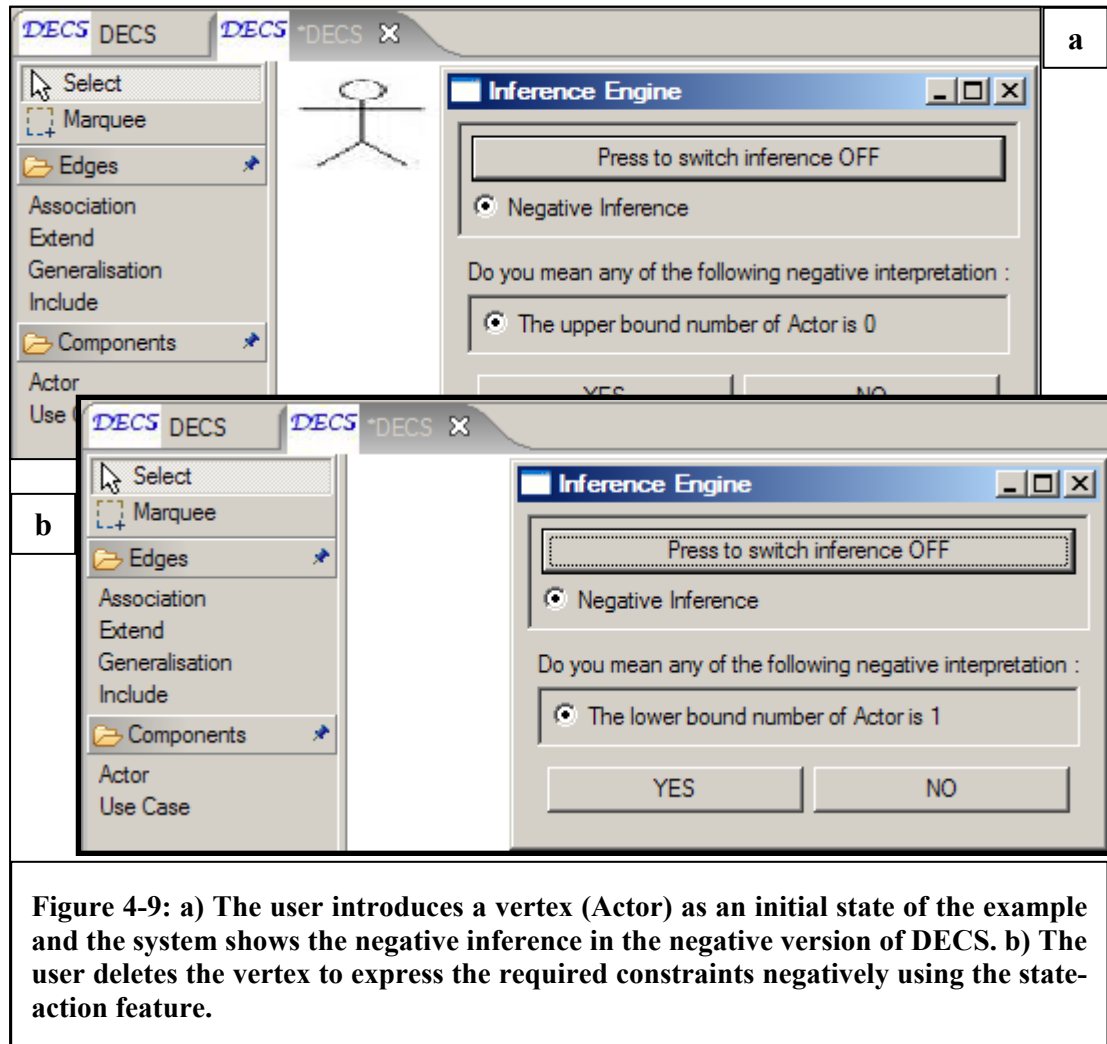


Figure 4-9 shows screenshots of a version of DECS that only depends on negative inference. However, it is supported with the state-action feature which allows the user to express the required constraint using a negative example. Figure 4-9-a shows part of the example which is composed of one Actor. Negatively, this is interpreted by DECS as the constraint “It is not allowed to have any Actor in the diagram.” (or as the figure shows “The upper bound number of Actor is 0.”). This part is considered as the state part of the example; however, the example has not been finished yet. The action part is when the user deletes the Actor vertex (Figure 4-9-b). From this action, the inference engine can infer that this action is not allowed. However, to be able to come up with reasonable constraint, the previous state of the

example must be taken into consideration. In this example, the previous state had one Actor and based on that the required constraint, “*It is not allowed to have less than one Actor in the diagram.*” (or as the figure shows “*The lower bound number of Actor is 1.*”) (or, “*There must be at least one Actor in the diagram.*”) will be inferred (Figure 4-9-b).

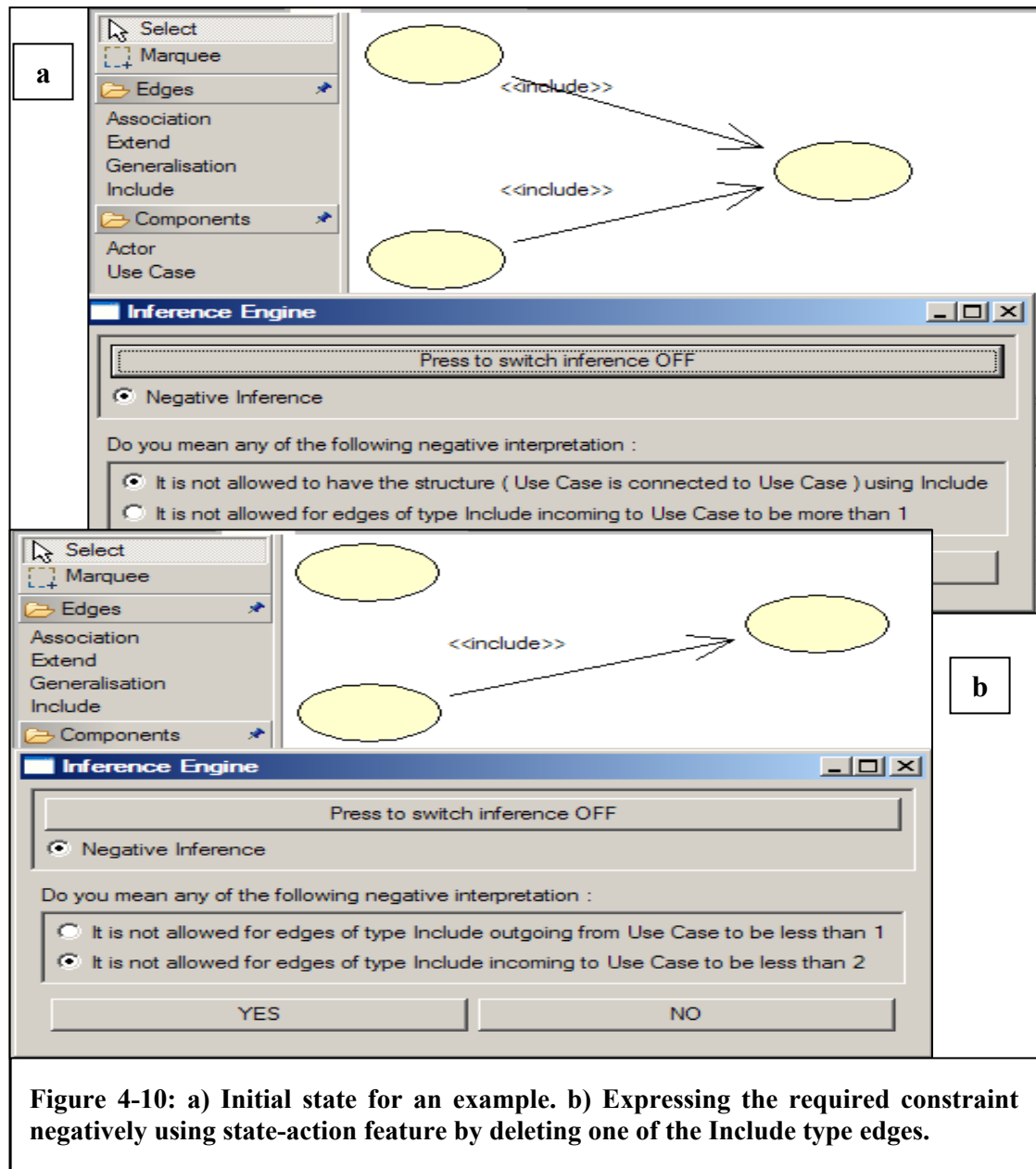


Figure 4-10: a) Initial state for an example. b) Expressing the required constraint negatively using state-action feature by deleting one of the Include type edges.

Because the state-action feature is an important part of the empirical study presented in 6.3, one more example will be presented. The example shows the process of expressing the constraint “*It is not allowed for edges of type Include incoming to Use Case to be less than 2.*”, (or “*It is not allowed to have less than 2*

edges of type *Include incoming to a Use Case*.”). The constraint is expressed using the state-action feature in Figure 4-10-(a and b).

The first step (Figure 4-10-a) shows the state of the constraint example which presents the required state (two Include edges are connected to a Use Case vertex as a target). However, because the system is interpreting the examples negatively, it does not infer the required constraint. Instead, it infers that the maximum allowed number of include edges incoming to the use case vertex is one. The next step of the example, the state-action part, requires deleting one of the include edges (Figure 4-10-b). This triggers the system to infer the required constraint as it recognises the previous graph state and the user action, which is undesired.

These examples show how the state-action feature enables the user to express examples using negative inference or interpretation. DECS is also able to infer constraints using positive example interpretation only with the support of state-action feature. It is possible to express the constraint “*It is not allowed to have less than one Actor in the diagram.*” (“*The lower bound number of Actor is 1*”) positively using an example with only one Actor as shown in Figure 4-11. However, the problem appears when trying to express, positively, the constraint “*It is not allowed to have more than one Actor in a Use Case diagram.*” (“*The upper bound number of Actor is 1.*”) or in the constraint “*It is not allowed to have more than one Start State in State Transition diagram*”.

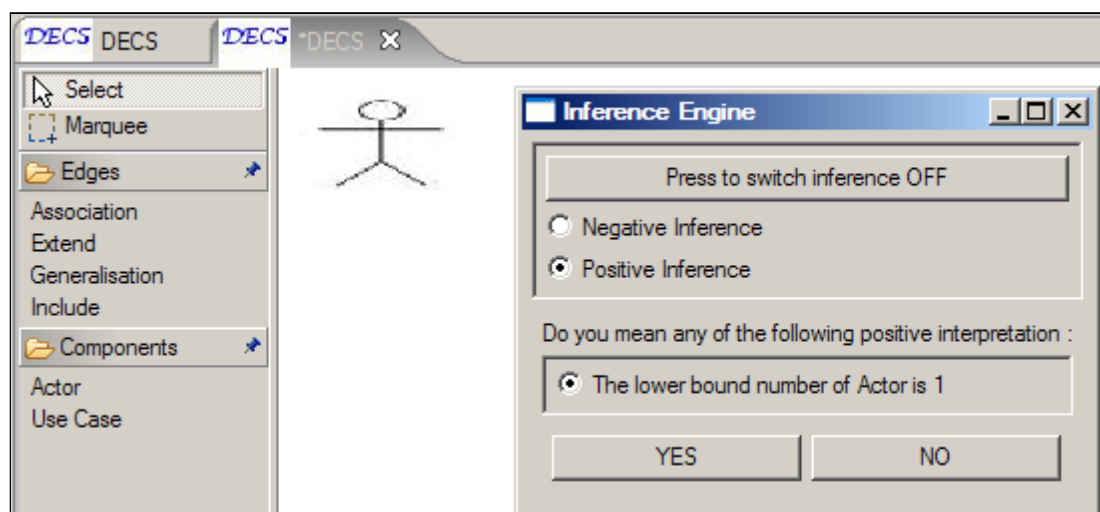


Figure 4-11: Positive interpretation (inference) the introduced example.

Both constraints reveal the same problem which is that these constraints cannot be expressed positively using a reasonable example. The only way of expressing the constraint is using the same example used for the lower bound number constraint. In other words, two contradicting constraints will be expressed using the same example which creates an arbitrary way of expressing the constraints that is undesired system behaviour. Although the last is more realistic and practical, for consistency with the above example, the first constraint will be used to explain the state-action feature.

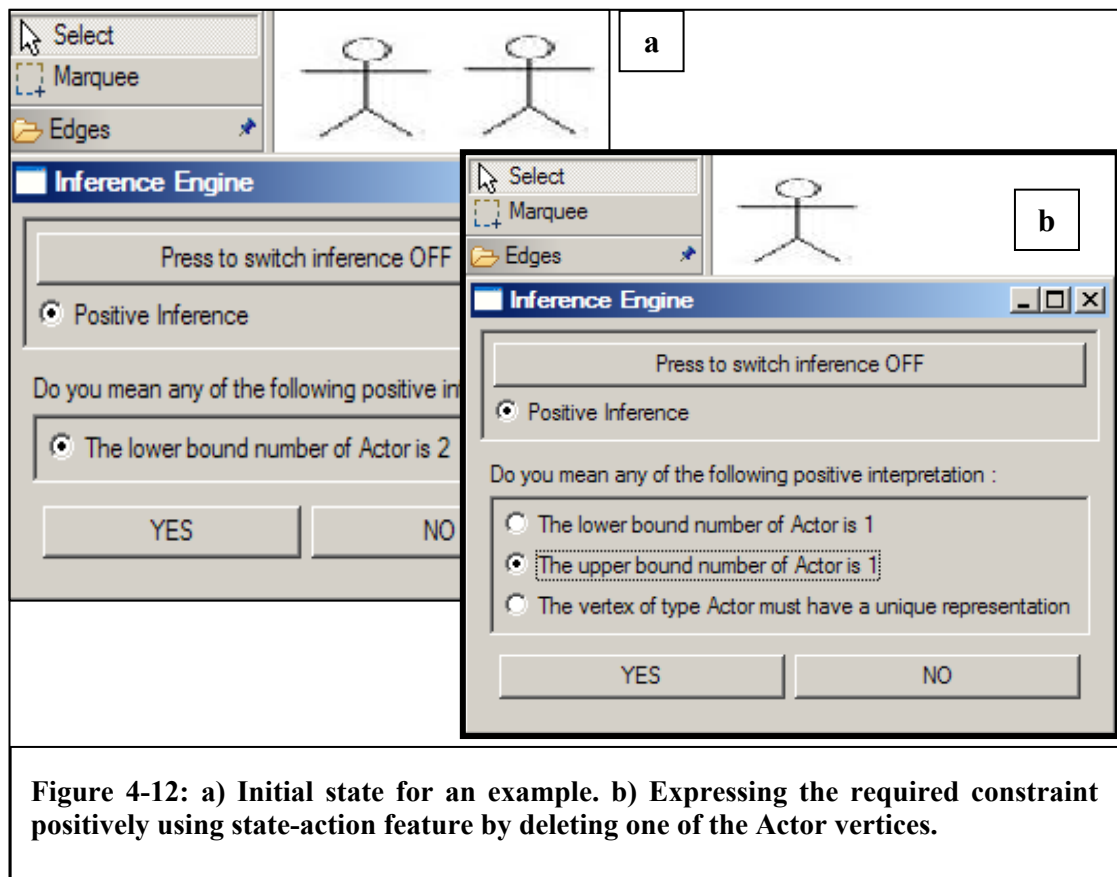


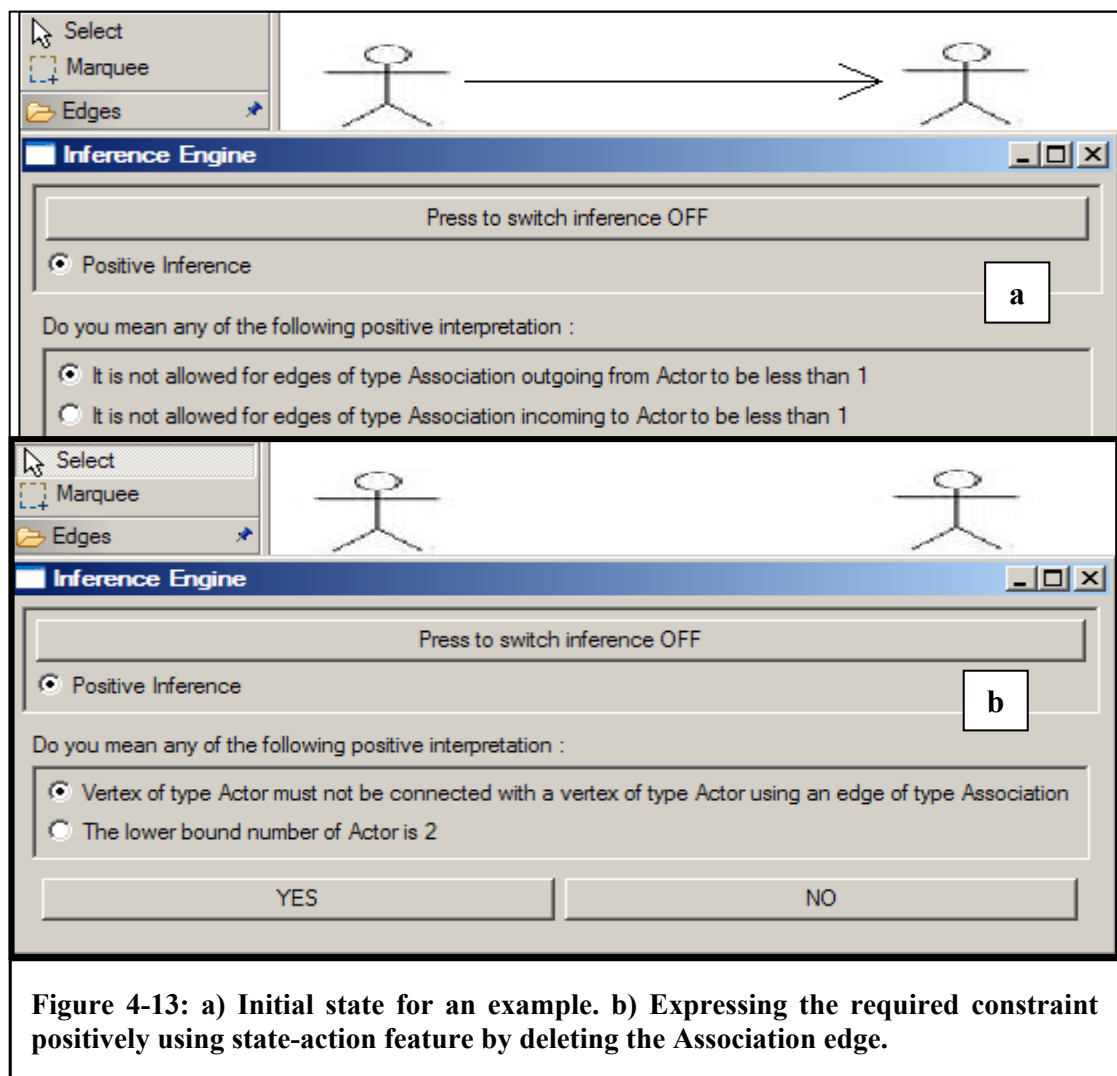
Figure 4-12: a) Initial state for an example. b) Expressing the required constraint positively using state-action feature by deleting one of the Actor vertices.

This constraint can be expressed, as shown in Figure 4-12-a and b, by introducing an example of two Actor vertices, which represents the state part of the example. The user then deletes one of them, which is the action part of the example. This gives a positive example for the required constraint. The only difference in this case is that the system will infer from the state that *follows* the action instead of the one that before it as in the case of negative examples. In the same way it is possible to define the connection constraint “A vertex of type Actor must not be connected to a vertex to type Actor using an edge of type Association”. Positive examples using the

state-action feature appear in Figure 4-12 and Figure 4-13 with a positive version of DECS.

There is a problem that appears in some cases in association with the state-action feature. This problem is clear in the constraint in Figure 4-13. When the user deletes one of the Actor vertices (Figure 4-12-b), the system generates more inferences than required because it infers based on the action and based on the state without the action. In Figure 4-12-b the system provides the inferences:

- The lower bound number of Actor is 1, and
- The upper bound number of Actor is 1.



The second is the required one which is generated as a result of the action; however, the first is inferred based on the state only. The problem here is that if the

system infers both constraints using the same example, what is the benefit from the action in this case? The user can only provide an example of one Actor only and both constraints are inferred without bothering with the action, which returns back to the same problem of providing extra arbitrary inferences from the example. This problem appears only in some cases, with the positive state-action examples only, because the system infers from the current state taking the action into consideration instead of the previous state as in the negative state-action examples. This problem does not appear in the second example (Figure 4-13). This problem could be solved by preventing the system from inferring from the current state only (without action consideration) when there *is* an action. However, this may create another difficulty which is that the user is not allowed to make any mistake or trials while introducing the example.

Expressing a constraint in a natural language and expressing the constraint in the system using an example are two completely different things. In a natural language, different designers might describe a constraint differently according to their way of expressing it when talking to each other. However, in DECS, the expression of the constraint is done by a diagram example that has no relation to how the designer expresses it linguistically. For example, in the above constraint, although it is expressed negatively in English by starting with words “it is not allowed...”, it is more natural to express it in DECS as a positive example. In English, this constraint can also be expressed positively as:

“At least two Actors must exist in the diagram”.

The example polarity for expressing a constraint has been studied in more detail throughout this research and an empirical study has been conducted to evaluate its effect. This experiment with another experiment conducted to evaluate the preference of expressing the constraints using natural language are documented in Chapter 6.

4.5.3 Inference Engine Transparency

To support the synergistic approach, DECS’ inference engine results are presented to the user in real time. When the user starts building an example (drawing a model), s/he can switch on the inference engine even before finishing the example.

This means that the user can watch and follow the inference engine at work while introducing the example(s). The following scenario clarifies this point.

The user wants to specify the constraint *“It is not allowed to connect Actor with Actor using Association edge”*. The user believes that he can express this constraint negatively by introducing the example: Actor is connected to Actor using Association edge as in Figure 4-8. The user switches the inference engine on before even start building the example. The inference engine has nothing (no inferences as no example provided). The user starts building the example by drawing (drags and drops) an Actor vertex. At this point the inference engine works and infers the constraints from the current state of the example which at this point is only one vertex of type Actor. This is presented in Figure 4-14.

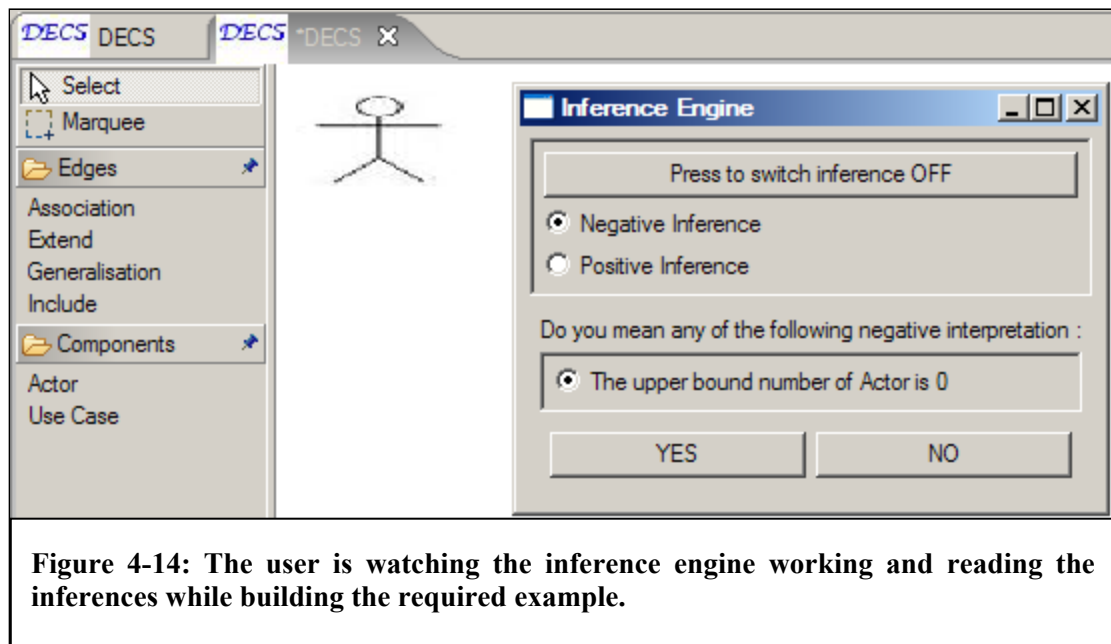
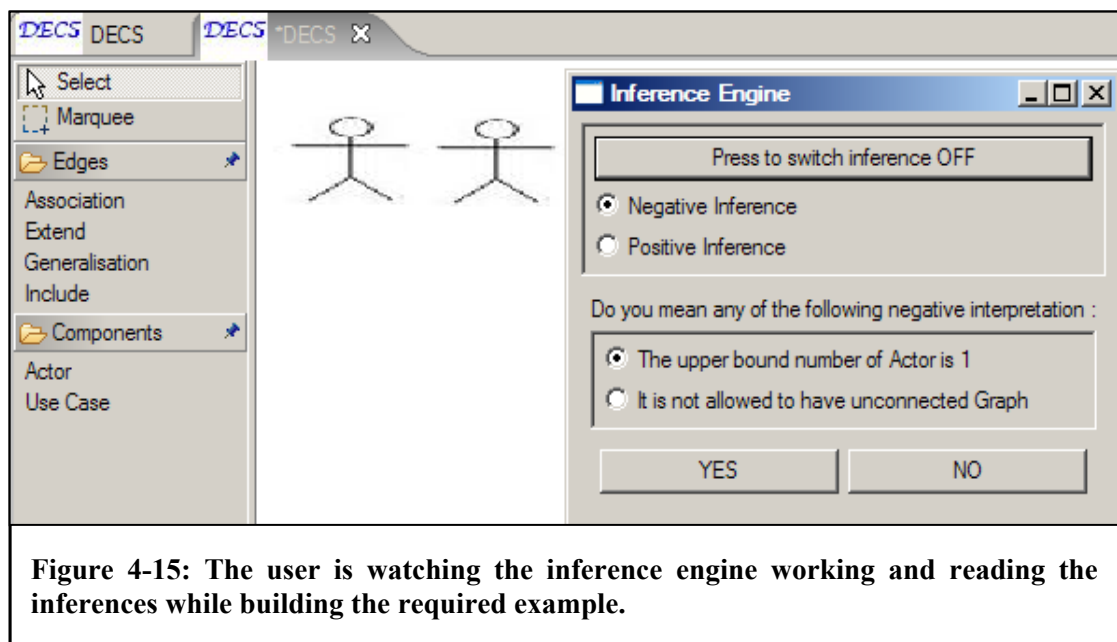


Figure 4-14: The user is watching the inference engine working and reading the inferences while building the required example.

The user reads the inferences before continuing the example. The user notices that the system infers a constraint that is related to cardinality. The user adds to the diagram another vertex of type Actor as in Figure 4-15.

The user notices again that the system infers a cardinality constraint and one more constraint from the example. Finally, the user connects the two vertices to achieve the required constraint. This step is shown in Figure 4-4. This scenario shows the meaning of inference engine transparency since the system reveals and

exposes the work of its inference engine while the user is building the required example.



This feature brings some advantages such as being able to track the development of the constraint step by step. If interested, the user can switch between positive and negative interpretations to look over and recognise the inferred constraints in every step. In the above scenario, the user understood that the system can infer cardinality constraints using example that have vertices. It is possible to argue here that this feature allows the user to understand the way that the inference engine works. It also helps the user to express his/her examples more easily. This can be clarified in case of the above scenario if the same user required later on to express a cardinality constraint on any vertex, he will be able to provide the required example easily as he has seen this before. If the user is not interested or bothered by continuous inferences, it is possible to stop the inference engine by pressing on “switch inference OFF” button. The user would need to switch it on again when the example is completed and inference is required.

A similar feature of transparency has been mentioned by Goldman & Balzer (1999) and called ‘synchronous analysis’. Although they did not implement it in their meta-tool, ISI, but suggested it as future work, they document that it would be a useful feature. They required it to show and update the feedback from the editor

concurrently with the work of the designer instead of showing the feedback when the designer requests this by pressing a button.

4.6 The Second Example: Synergistic Approach and Example Remodelling (Visual Generalisation)

In the previous example, it was possible to express the required constraint using only one example. This is difficult in some other cases as the following example shows. If the user wants to specify the constraint:

“It is not allowed to connect a vertex of type Use Case (as a source) to a vertex of type Actor (as a target) using any edge type”.

There are two ways to express this constraint and both show a novel feature of DECS. The first is initiated by the user since s/he introduces four different examples, all of them showing Use Case as a source and Actor as a target and different edge types in each example (Figure 4-16-a).

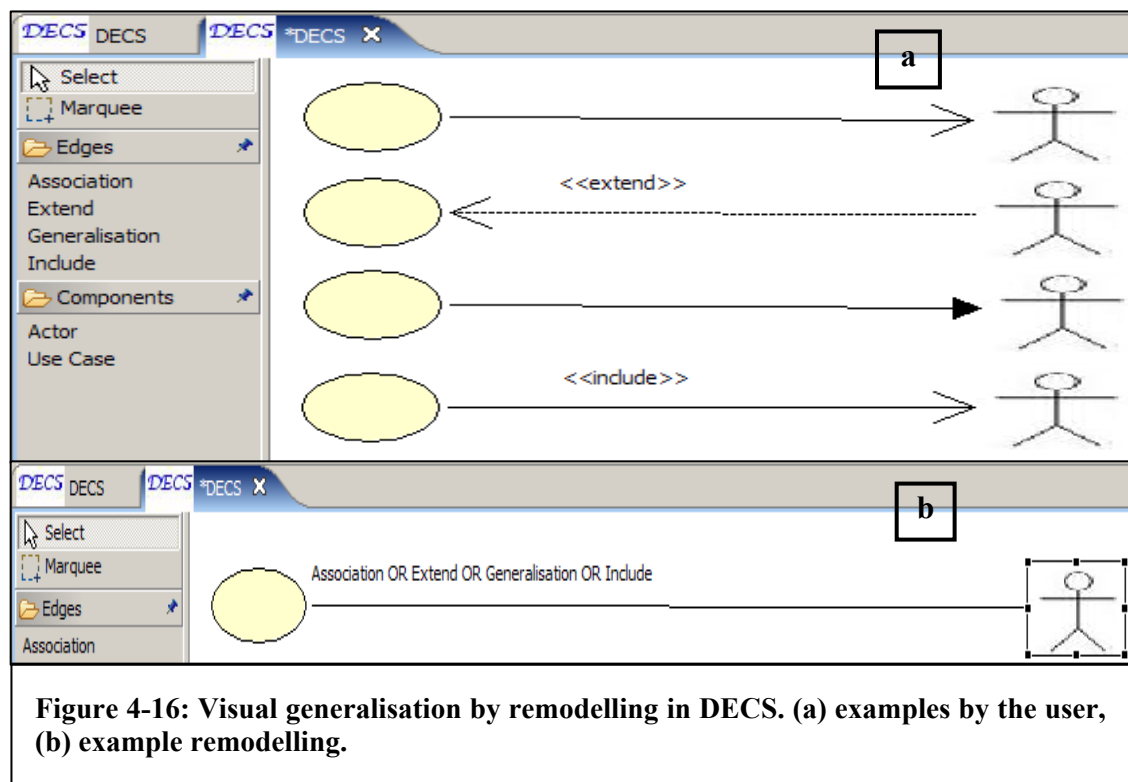
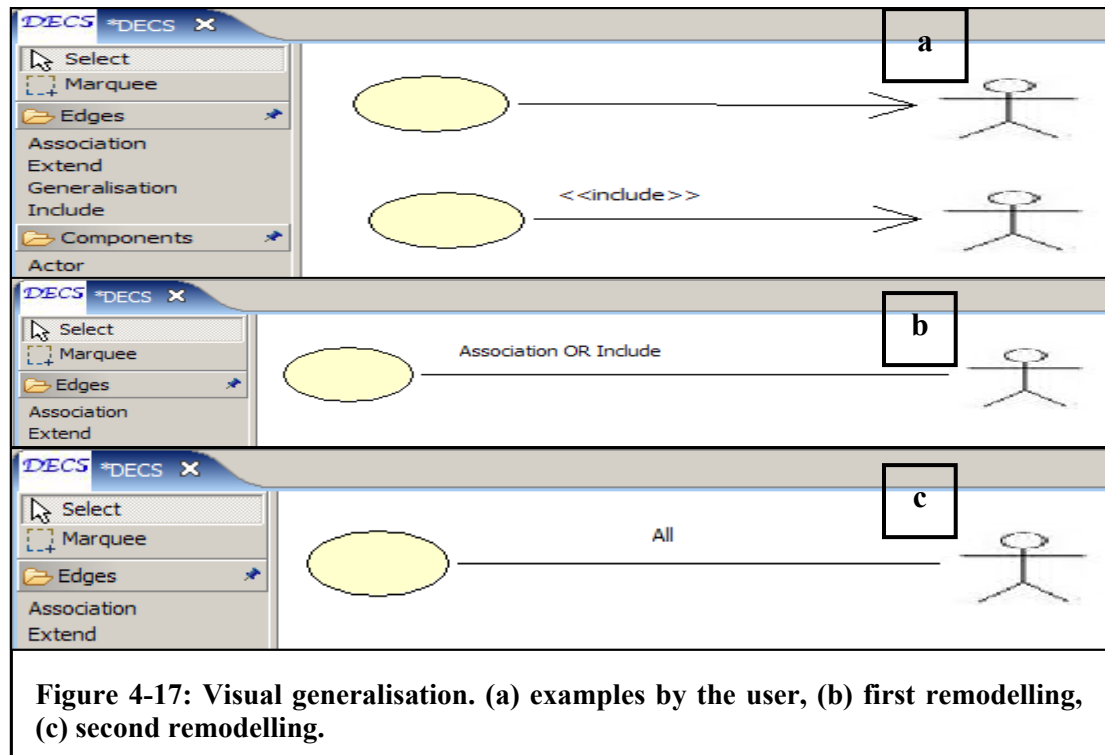


Figure 4-16: Visual generalisation by remodelling in DECS. (a) examples by the user, (b) example remodelling.

The system cannot directly jump to the conclusion that the diagram represents four examples for the same constraint; instead it assumes that it is only one example. Accordingly, it infers that these four structures must not be allowed together in a diagram and shows this interpretation to the user. Since this is not the intended constraint, the user presses “NO” to tell the system that the required constraint is not in the inferred list. The system goes a step further by decomposing what has been considered as one example into different examples expressing one constraint. Therefore, the system generalises the examples visually by joining (fusing) them together into one example. In this research, this process is called ‘visual generalisation’. In DECS visual generalisation is achieved by remodelling. The edge in the new example will be represented by a general edge type with a label showing the generalised edge types (see Figure 4-16-b where the label says Association OR Extend OR Generalisation OR Include). The remodelled example is used by the system in another inference attempt as shown in the model of the synergistic approach in CSBE (Figure 4-1). In other words, the system considers the remodelled example as an example introduced by the user and makes inferences based on it. Consequently, the previously inferred constraint list is updated, showing the interpretations of the newly remodelled example. This ends the system turn and the control returns back to the user to select from the list. Since the newly inferred constraint list contains the intended constraint, the user selects it and presses “YES”. The process of naming and constraint generation continues as in the first example.

In this scenario the user introduced four examples to express the constraint. However, because the intended constraint contains all the edge types (the four edge types defined in the Use Case diagram), the same constraint can be expressed in an easier way that depends on the generalisation abilities of the CSBE technique in DECS.

The user can introduce only two examples showing a Use Case connected to an Actor using any of the four edge types, say Association and Include for this scenario (Figure 4-17-a).



The system again considers this as one example and shows inferences based on that. The user rejects the inference. The system decomposes the example into different structures and considers every structure as a separate example. Consequently, the system generalises the two examples, generated from the decomposition, by fusion and generates one example showing the fused edge with a label indicating the Association and Include edge types only (Figure 4-17-b). The system infers again based on the remodelled example and the inferred list will contain a constraint including only Association and Include edge types. Until now the behaviour is exactly similar to the last scenario. However, in this scenario, the user again rejects the inference since the intended constraint is not in the list. The system recognises that the user introduced two different edges from the diagram's available edge types. Based on that, it generalises to include all the available edge types in the example. To generalise the example visually, the system changes the label of the edge to "All" meaning that all diagram edge types are included and the inference list is updated to contain the required constraint this time (Figure 4-17-c).

Generalisation in DECS is distinctive because it can be considered as visual generalisation. As Figure 4-16 and Figure 4-17 show, DECS generalises an example from the initially introduced examples and presents it visually to the user. Although

the term ‘visual generalisation’ has been used before in PBE systems (Amant, Lieberman, Potter, & Zettlemoyer, 2000), it was not used to refer to the same concept introduced in this dissertation. Their concept of visual generalisation is the ability of a PBE tool to generalise and learn from user behaviour based on the visual properties of objects, such as the colour of a hyperlink in a web browser, instead of depending on the application data model, such as depending circles and boxes in a drawing application. The concept and the implementation of visual generalisation which includes remodelling and depends on the generalised object as an input again for the inference process, has not been implemented in the same way before in any reviewed PBE tool.

4.7 Inference Engine Rules

The inference process in DECS depends on a rule-based inference engine. Rules are expressed as IF-THEN statements, stored as text and converted to objects at runtime. Each rule consists of two parts, an IF part and a THEN part each represented by a runtime object. The IF part tests whether or not the condition is satisfied. The conditions tested in this part of the rules are diagram features that must be satisfied by the model (the example).

If the IF part is satisfied, the THEN part is executed to generate a string describing the inferred constraint. These constraint descriptions are what is presented to the user in the inferred constraints list. In other words, each rule is responsible for inferring only one constraint. When the user selects an inference that represents the required constraint from the list, the rule that inferred the selected constraint description will be responsible for generating the constraint code (XML script). The THEN Java class performs this task.

The DECS inference engine has two types of rules called ‘*choice rules*’ and ‘*remodelling rules*’. Choice rules are responsible for generating the inference list presented to the user such as those presented in the previous screenshots (Figure 4-5-b) as an example. Remodelling rules are responsible for the visual generalisation and example remodelling introduced above (Figure 4-15 and Figure 4-16). These rules perform the action of modifying the example by fusing the elements and generating

the appropriate labels. To connect the CSBE synergistic process ideas together, choice rules work while the user introduces examples and remodelling rules work when the user cannot find the intended constraint in the list and rejects all the inferred constraints.

Both rule types depend on extracting features from the example(s). Choice rules are interested in features such as the connection of the vertices, the cardinality of connections, the cardinality of vertices of the same type, and similarities between labels. Remodelling rules extract features that are used to compare the introduced examples. They check similarities and differences between the properties such as the types of source vertices, types of target vertices, and types of edges. These rules allow the user to express the important parts of the constraint (the example parts that are required to be involved in the constraint). In the last two examples, remodelling rules were used to infer that the user is not interested in generalising the source vertex or target vertices because they are the same in all examples. By contrast, the user shows the required part to be generalised by making it different in the different examples. If the user wants to generalise vertex types, s/he should introduce examples of different vertex types. The system will generalise and remodel exactly as in the case of edges. In the case of vertices, the system will model different vertex types by a rectangle with label showing the vertex types in the remodelling and “All” in the generalisation as shown in Figure 4-17.

The list of features that DECS depends on for inference includes features such as the number of structures and the number of vertices in the model. DECS depends on three different types of features that are related to the different types of rules. The first type of features, “the state inference features”, is used to infer according to the state of the model. This type is used by the choice rules. The second type, “the action inference features”, is for inference based on the state-action rules which are also considered as a sub-category of choice rules as they generate choices, albeit with action. The third type, “visual generalisation inference features”, helps to infer the opportunity for visual generalisation (remodelling) and, accordingly, are used by the remodelling rules. A full list of feature sets is given in Appendix F Section F.1. A fourth type of feature is used by DECS but for the purpose of generalisation instead of inference in inference rule augmentation and learning (discussed in the next

paragraph). All the types of features that DECS depends on either for inference or for learning mechanism are fixed. They are only extendable by direct programming. This is considered as a limitation DECS; however, it offers a future research opportunity.

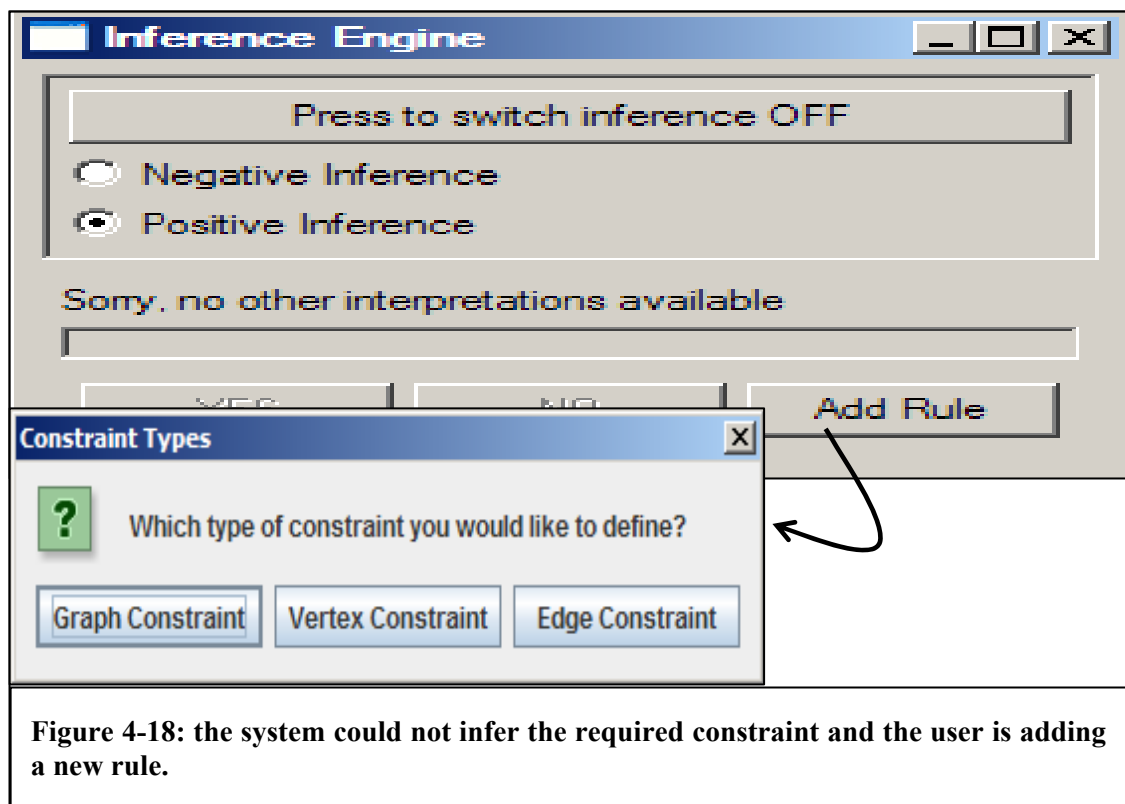
4.8 Inference Engine Augmentation and Learning

DECS, like other rule-based PBE systems, has a fixed rule set that has been implemented by the designer for the inference, remodelling and generalisation processes. In a context such as software engineering constraints, this is typically not sufficient, since it is unlikely that a complete set of rules can be known and implemented a priori. As a solution, rule learning has been implemented in the DECS' CSBE technique as a part of the synergistic interaction between the system and the user (recall Figure 4-1 Page 97). The learning technique has not been implemented before in any rule-based PBE system. The technique depends on the user teaching the system while s/he is working on constraint specification. This technique will be introduced here through the third scenario which is based on the following constraint:

“It is not allowed to connect two vertices of type Actor using Association edge”.

It is possible to define this constraint negatively as in the first example; however, we will assume that the user has chosen to provide a positive example to express this constraint. Thus, the user provides an example showing two Actors with no connections between them to express a positive example for the constraint. The user asks the system to interpret the example and selects positive interpretation radio button. The user does not find the intended constraint in the generated list, so presses “NO” button. The system switches to “remodelling” rules trying to remodel or generalise but no rules are triggered. The system shows an apology message telling the user that the inference engine does not have the required knowledge to infer the intended constraint from the example. This is not the end of the story; the system offers the user the option to add a new rule (recall Figure 4-1 Page 97). Adding a new rule is considered a system learning mechanism since it allows the system to

recognise the example in the future and to infer the required constraint. The user presses on the “Add Rule” button (Figure 4-18).



Although this process is called adding rule, it is actually a constraint specification process and the system learns from it how to specify the constraint using the example. In other words, the user will not only add a rule, but also will define the required constraint (using DECS' constraint definition wizard).

The system asks the user to choose the required constraint type: graph, vertex, or edge. The user chooses by pressing on one of the buttons (Figure 4-18). At this point DECS transfers to a “form-filling” technique. Similar switching from one constraint specification technique to another has been introduced in (Druid) (Singh, Kok, & Ngan, 1990). At this point DECS calls the required form based on the choice of the user. The user fills the form as required and saves the constraint. The system at this stage has another activity which is learning from the specified constraint. It generates a file that holds a map between the introduced example and the generated constraint. In the future, when the user introduces the same example, the newly defined constraint description will appear in the list. In this way, the user specifies the

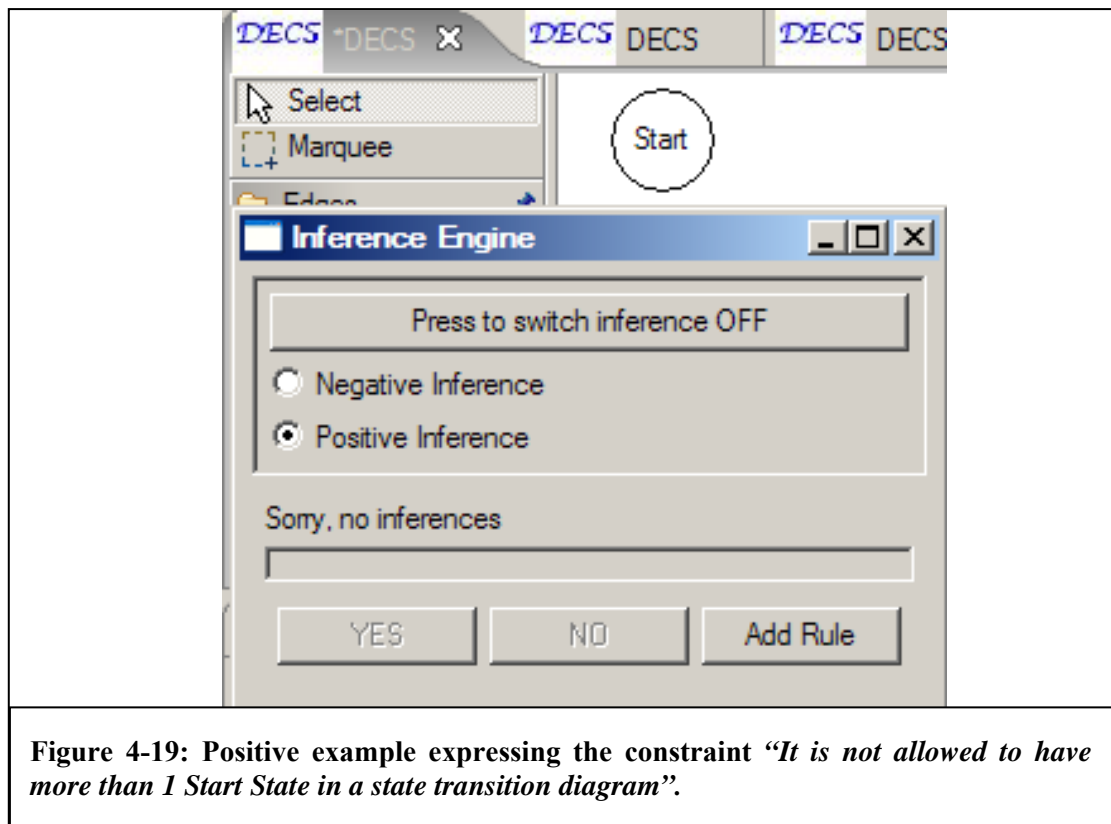
required constraint and, at the same time, teaches the system how to recognise examples of it.

This scenario explains how to add a rule using a new or customised example. However, the scenario only presents how to use the added rule (the taught example) based on presenting exactly the same example again. This shows a deficiency in the system learning technique. This deficiency can be clarified in the following scenario. The scenario also clarifies the ability of customising and adapting DECS to infer constraints using a customised example of the user's choice instead of the already implemented ones:

The user is trying to define the constraint:

“It is not allowed to have more than 1 Start State in a state transition diagram”

This constraint can be defined using a negative example by providing two start states. However, the user provides only one start state as an example and they believe that this is a convenient positive example to express this constraint (Figure 4-19).



The system does not infer the required constraint from the introduced example because the system inference engine does not have the knowledge to infer the required constraint from the introduced example.

The user uses the “add rule” feature to teach the system how to define the constraint represented by this example. This is done by pressing the “Add Rule” button which starts the process of adding a rule and teaching the system how to define the constraint. The process starts by asking the user to select the required constraint type (Figure 4-18).

Each of the three alternatives (Graph Constraint, Vertex Constraint, or Edge Constraint) will result in a different form to fill in to specify the constraint. For the current example the “Graph Constraint” type is selected which results in showing the form required to specify a graph constraint (Figure 4-20).

The screenshot shows a software interface with a sidebar on the left containing a 'Start' button and a 'Properties' tab. The main window is titled 'Edge Properties' and 'Vertex Properties'. The 'Edge Properties' tab is selected. It contains the following fields and controls:

- Constraint Name and Type:**
 - Constraint Name: [Text Field]
 - Prevent Cyclic Graph: ☐
 - Polarity: [Positive] (dropdown)
 - Constraint Type: [hard] (dropdown)
 - Location: [Text Field] with a [Browse] button.
- Edges:**
 - Lower Bound Number:**
 - URI: [Text Field] with [Browse] and [New] buttons.
 - Value: [Text Field] with [ADD] and [Remove] buttons.
 - Logic buttons: [OR], [AND], [COM] and a closing parenthesis [)].
 - Upper Bound Number:**
 - URI: [Text Field] with [Browse] and [New] buttons.
 - Value: [Text Field] with [ADD] and [Remove] buttons.
 - Logic buttons: [OR], [AND], [COM] and a closing parenthesis [)].
- Bottom Section:**
 - URI: [Text Field] with [Browse] and [New] buttons.
 - Unlabelled Edges: ☐
 - Identical Labels: ☐
 - Identical In-Edge Label: ☐
 - Identical Out-Edge Labels: ☐

Figure 4-20: The system provides the user with a form to define a graph constraint.

This form is filled as shown in Figure 4-21 which shows only part of the form. Figure 4-21 shows that the user has entered the constraint name “UBN1” (to represent ‘Upper Bound Number is 1’), the constraint type is “hard” and the location of saving the constraint is URI “D:\”. The user also specifies the polarity property of the constraint as “Positive” which specifies that the constraint appears in the positive list (or when the positive interpretation of the example is required).

The screenshot shows a window titled 'Vertex Properties' with a tab labeled 'Vertex Properties'. Inside, there's a section 'Constraint Name and Type'. It contains a text field for 'Constraint Name' with the value 'UBN1', a checkbox for 'Prevent Cyclic Graph' which is unchecked, a dropdown for 'Polarity' set to 'Positive', another dropdown for 'Constraint Type' set to 'hard', and a text field for 'Location' with the value 'D:\'. A 'Browse' button is located to the right of the 'Location' field.

Figure 4-21: The user enters some properties as a first step.

Since the constraint is related to the vertices, the user switches to the “Vertex Properties” tab in the form above which results in presenting the form in Figure 4-22-a. The user fills in the required parts of the form by entering the description, which is the message that will appear to the user when the constraint is violated. The user then enters the value of the upper bound number which is 1 and enters the URI for the vertex, in this case of type Start State. Since the user has not already defined such a constraint, they press the “New” button. A vertex constraint form (out of the graph constraint form currently in use) (Figure 4-22-b) is presented and the user enters the required values which in this case are the constraint name “BasicStartState”, the location to save “D:\”, and the vertex type “Start State”. The form contains other properties such as the polarity and the constraint type but they are ignored by the system as they are not relevant to the specification of the current constraint. The user saves the “BasicStartState” constraint which returns the process again to the main constraint. The reference URI appears in the URI text field. The user presses the button “Add” to add the URI to the text area as appears in (Figure 4-22-a). The user finishes the process by saving the constraint.

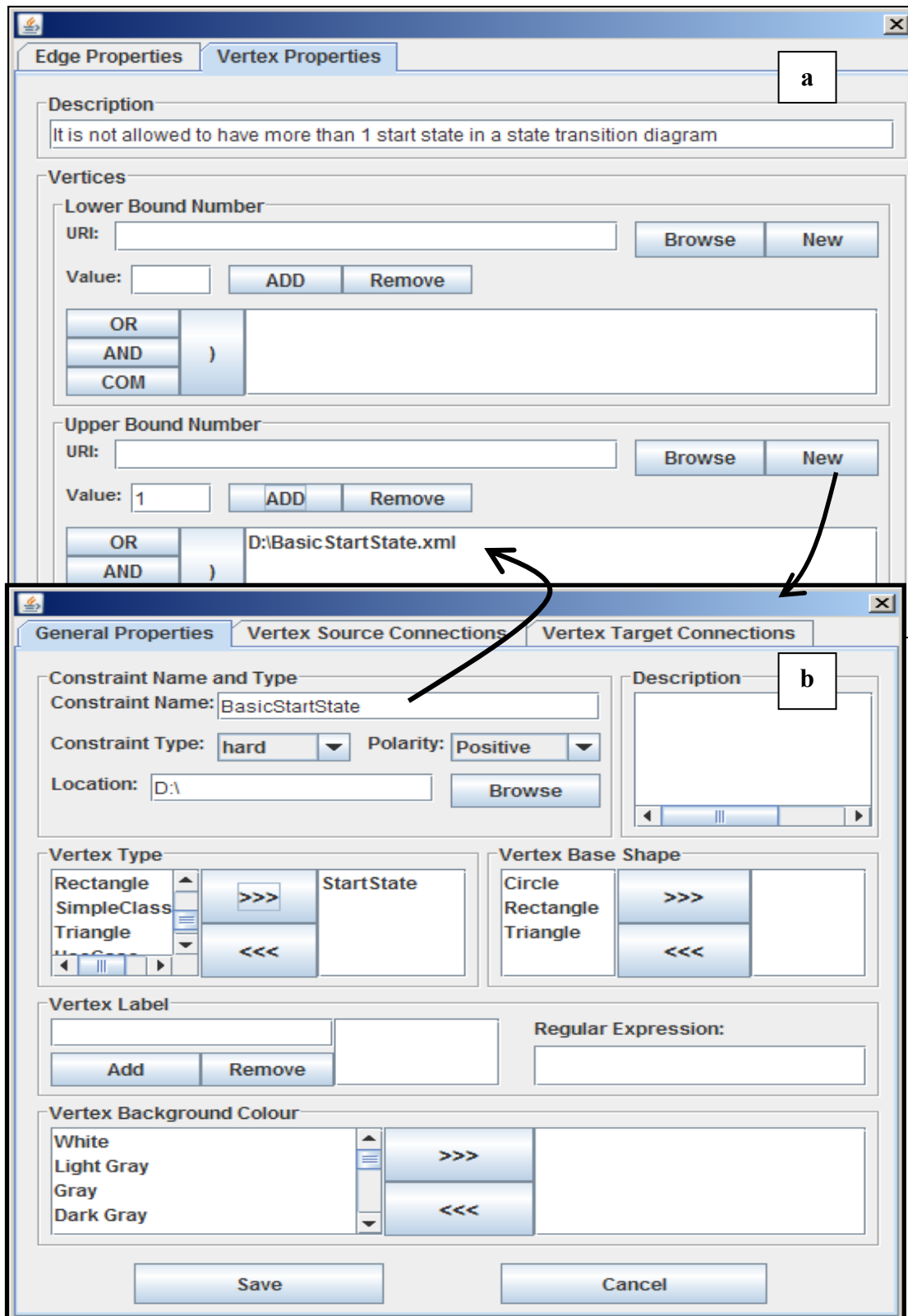
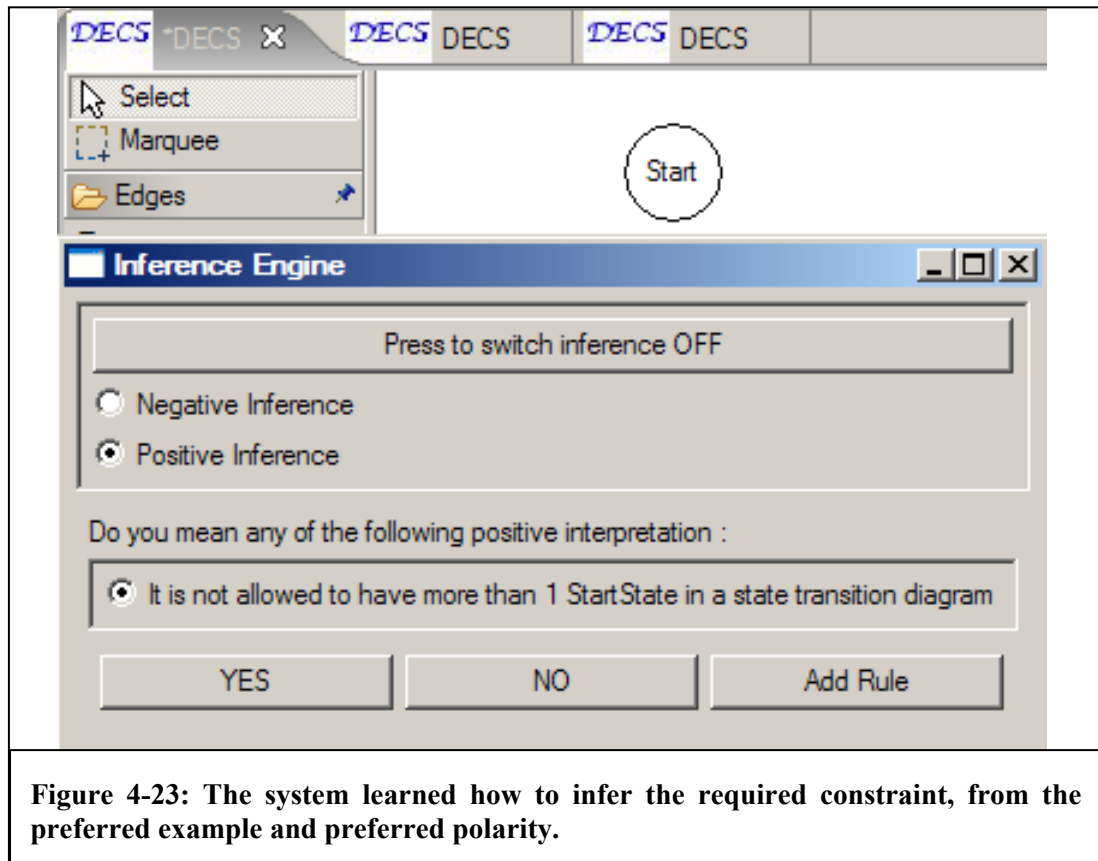


Figure 4-22: a) Continue specifying the constraint UBN 1. b) Specification of the Start State as a separate constraint that is referenced from UBN 1.

Once the user saves the constraint, the constraint is completely specified using the form-filling technique. At the same time, the system learned how to define the constraint using the example shown in Figure 4-19.

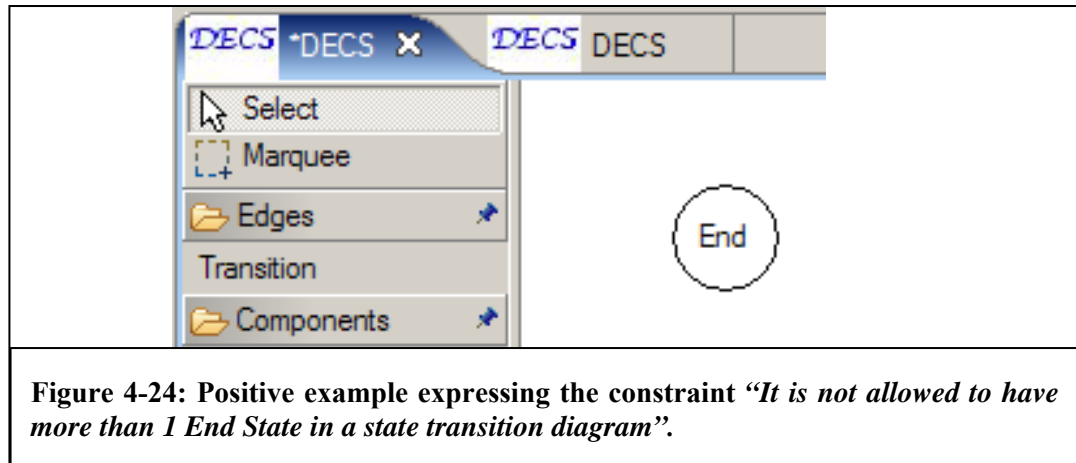
In a later session, the user requires to define the same constraint. The user uses the same positive example that the system trained on and the result is in Figure 4-23.



The user also wants, either in the same session, or in a later one, to define the constraint:

“It is not allowed to have more than 1 End State in a state transition diagram.”

If the user expressed this constraint in the same way as they did in the previous example, the user will express the constraint using one End State as they previously expressed the Start State constraint. The example is shown in Figure 4-24.



The user needs to repeat the whole process above since the system cannot recognise the similarities between the current example (one End State) and the previous example (one Start State). This is because the inference manager searches for an identical example instead of searching for a similar one. In other words, the system is not able to generalise the first example so that the second example is recognised as of the same type.

4.8.1 Generalisation

To enhance the learning technique and acquire the full benefit of it, a generalisation feature has been implemented in DECS as a component called the “adding rule manager”. Essentially, this generalisation feature is a form of inference from an example that reduces the number of situations in which learning must be applied. The following algorithm describes the implementation of this feature and the previous example is used to explain it:

Step 1: The first step is to recognise the similarities between the learned example and the new example. Accordingly, similarity checking has been implemented instead of searching for identical examples only. When the user provides an example for the first time to define a constraint and the system does not have the knowledge to infer the required constraint, the user uses the “Add Rule” feature to define the constraint using the wizard. The user saves the specified constraint. At this point, the ‘inference manager’ sends the set of features that were satisfied (triggered) as a result of the provided example to the ‘adding rule manager’. These features are those that are normally satisfied when an example is introduced

and the ‘inference manager’ uses them to trigger the state and the action rules discussed in Chapter 4. These features are available in Appendix F Sections F.1.1 (state inference features) and F.1.2 (action inference features). In the case of the above example, when the user provided the Start State example, only one feature is satisfied which is “the graph has a single vertex”. The ‘adding rule manager’ receives the set of satisfied features and adds them as a script to an XML file that contains and represents the newly added rules. This XML file will contain information that the ‘adding rule manager’ receives about the introduced example and this is considered as the “added rule”. This information can be listed as:

- The set of satisfied features (explained above).
- The example itself (the picture) is saved as an object in a file and its URI reference is added to the XML ‘added rules’ file.
- The textual description introduced by the user which describes the constraint (see the ‘Description’ field in Figure 4-22-a).
- The constraint itself is saved as an object in a file and its URI reference is added to the XML ‘added rules’ file.

The above information is considered as an added rule. Later on, it will be clearer that the IF part of this rule is the set of satisfied features and the THEN part is the constraint textual description. Figure 4-25 depicts the ‘added rules’ XML file with the added rule above.

```
<?xml version="1.0" encoding="UTF-8"?>
<AddedRules>
  <Rule>
    <tr>if graph contains GSingleS</tr>
    <serialized>H:\addedRules\SerializedFiles\serialized2.dat</serialized>
    <constraint>H:\addedRules\SerializedFiles\UBN1.xml</constraint>
    <description>It is not allowed to have more than 1
      StartState in a State Transition Diagram</description>
    <polarity>p</polarity>
  </Rule>
  <Rule>
    .....some other Added Rule
  </Rule>
</AddedRules>
```

Figure 4-25: The ‘added rules’ XML file

Before going further, assume that in a later session, the user introduced an identical example (one Start State) again. In this case, the system can recognise the new example as being identical to the previous one using the saved original example picture. However, assume that the new example is similar, instead of identical, such as the case of the ‘End State’ example above. In this case, the following happen:

- Some features in the ‘inference manager’ will be satisfied; in this case “the graph has a single vertex” will be the only one.
- None of the original rules in the ‘inference manager’ will be triggered like in the first example.
- The ‘inference manager’ sends the triggered features to the ‘adding rule manager’ which will search the added rules XML file.
- The manager will compare the newly received satisfied features with the features already saved with each added rule.
- The manager will discover a match for features (in this case one feature) with the ‘Start State’ added rule.

In other words, the system generalises the already saved examples at learning time and searches for similar examples to the currently provided example instead of searching only for identical ones. This allows the system to recognise similar examples.

Step 2: Step 1 solves the problem of recognising similarities between examples but does not offer a complete solution to the problem of generalisation. To be able to generalise more fully, the system retrieves the previously saved example (the object file) and analyses it for opportunities to generalise properties in the constraint descriptions (the textual description). This is achieved via the following scenario:

- The ‘adding rule manager’ converts the original example file using its available URI into a Java object.
- It compares the elements (vertices and edges) in the original example with the elements in the new example. To be able to generalise, the comparison depends on some features called “rule generalisation features” (see Appendix F Section F.1.4). In case of ‘Start State’ and ‘End State’ example, the manager retrieves the

“Start State” example and compares it with the “End State” example. The system checks the “Start State” vertex and collects the features

- “vertex type = “Start State”,
 - “incoming edges = 0” and
 - “outgoing edges = 0”.
- The system searches in the new example for a vertex with the same features but it fails to find the same vertex type. This is the case where the system tries to find the identical example. However, the system finds the “End State” which satisfies the final two features from the first example. In this case the manager considers this as a match.
- To be able to know what property should be generalised, the system retrieves the constraint object to check the properties that were specified in the original constraint. In this case, the only specified property in the original example was the vertex type ‘Start State’. Note that selecting the property to generalise does not have any relation with any particular type of features because the features are only for discovering a match. After the match is discovered, the generalisation takes place based on the previously specified properties in the original constraint. However, the only common property between the features and the generalisation is the element type because the process depends mainly on it. This is considered a limitation in DECS that should be addressed but it was not necessary during the empirical studies to generalise over properties apart from the element type. By contrast, all the required constraints were generalised over the type in addition to other properties. In addition to the element type, DECS generalises the colours, the labels and vertex cardinality.
- The task now is to generalise the vertex type by replacing the ‘Start State’ with ‘End State’ strings in the original constraint textual description available in the added rule. This is done by parsing and replacement of the textual description. The description becomes “*It is not allowed to have more than 1 End State in a state transition diagram*”. Another limitation in the generalisation process lies in the parsing. This because there may be spelling mistakes in the original text or the user might not have included the property name in the text; in these cases, ,

the text will not be modified⁸. However, the new constraint generation (Step 3) will not be affected.

- The end of this step is that the modified textual description is returned back from the ‘adding rule manager’ to the ‘inference manager’ which adds it to the constraint list that will be presented to the user under the suitable interpretation polarity (positive in the current example as appears in the rule XML file). This is shown in Figure 4-26 since the user provides End State as an example, and the system generalises from the previously learned example.

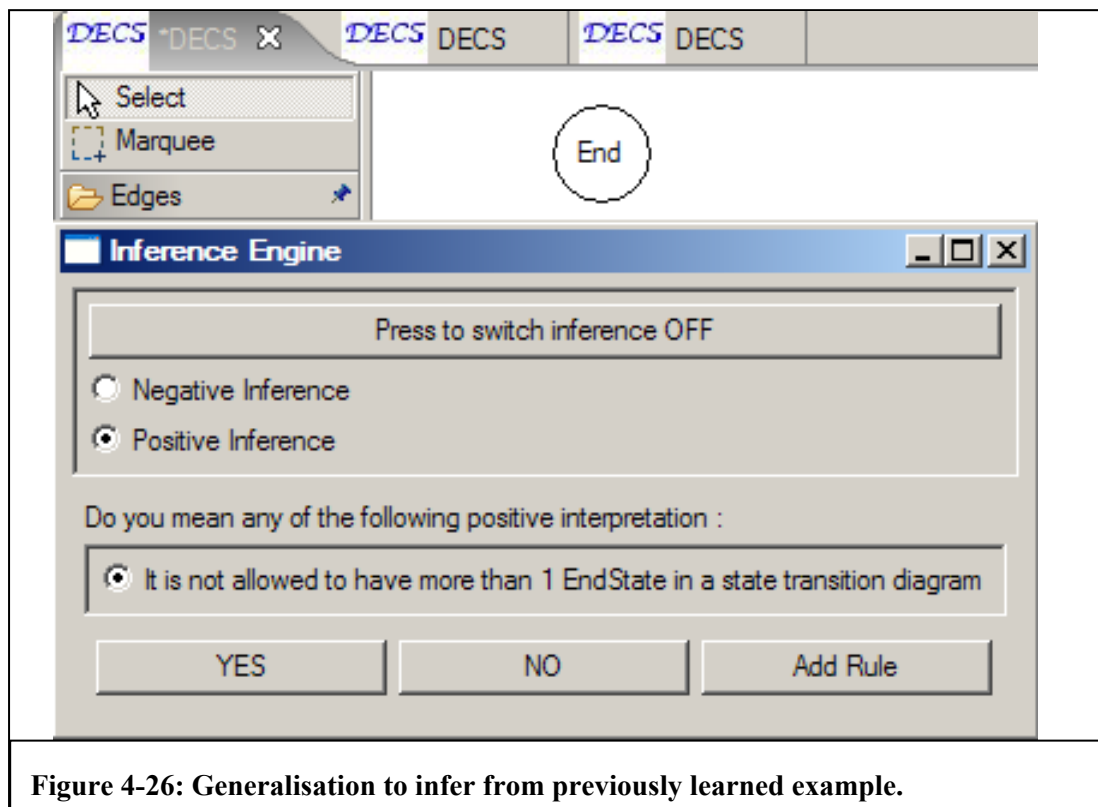


Figure 4-26: Generalisation to infer from previously learned example.

⁸ Note that the term ‘Start State’ in the string must be identical to the value of the vertex type property in the constraint definition. Currently, this restriction is not maintained by DECS itself; the person who defines the constraint must make sure that this equivalence is satisfied.

Step 3: The inferred constraint list is presented to the user. If the user selects the constraint description generated from an added rule and presses ‘OK’ to confirm that this is the required constraint, the following happens:

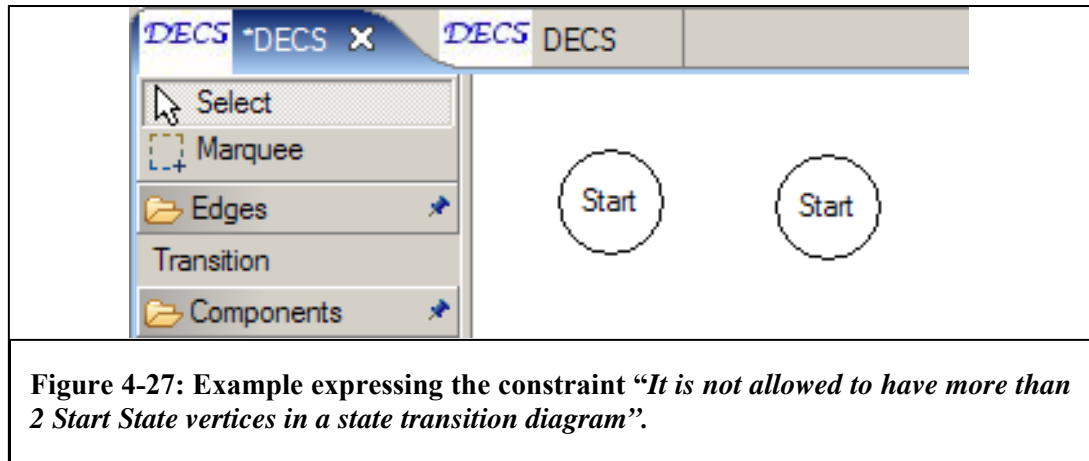
- A dialog asks the user to enter a name (a location) for the new constraint.
- The ‘inference manager’ knows that the selected constraint textual description is generated from an added rule, so it delegates the task of creating the constraint object to the ‘adding rule manager’ and sends it the new constraint name and location.
- The ‘adding rule manager’, again, retrieves the original constraint object.
- The manager makes a copy of the original constraint object and performs another generalisation process that is exactly the same as the previous one done in Step 2. However, this time the generalisation replacement is performed over the properties values in the new constraint object.
- The new constraint is generated in the specified location.

4.8.2 Some Complicated Scenarios

The above scenario is the simplest possible generalisation scenario with only one feature required to be generalised, viz., vertex type. Here are some complicated examples (involve generalising properties over more than one component in the example) that show the advantages and limitations of the “adding rule” feature. This feature is able to generalise the number of vertices or edges if they are more than one. The following constraints clarify the idea:

“It is not allowed to have more than 2 Start State vertices in a state transition diagram.”

Assume the user expresses this constraint positively using an example consisting of two Start State vertices (Figure 4-27).

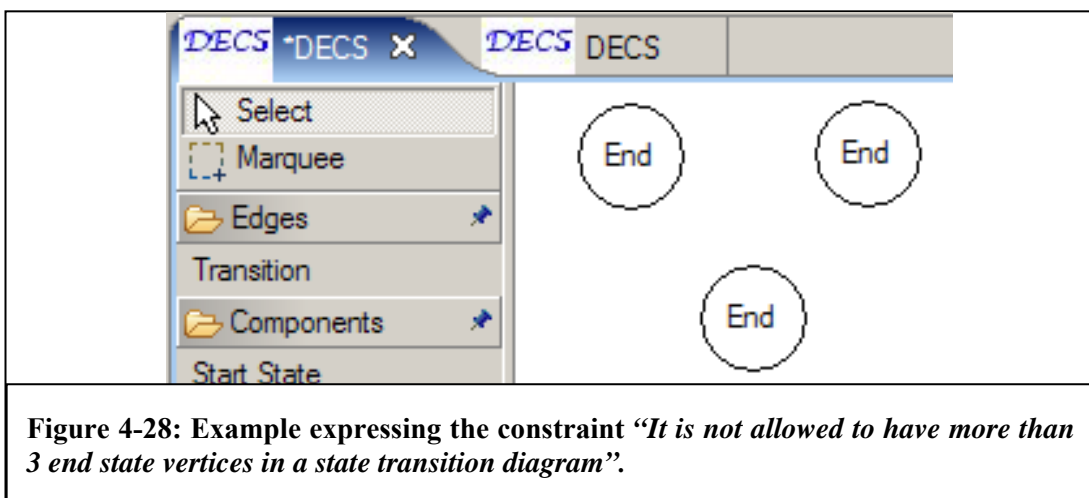


Assume also that the inference engine does not have the required knowledge to infer this constraint using the example provided. In this case, the user uses the “adding rule” feature to teach the system to specify the constraint as the upper bound number of start state vertices is two. The triggered feature in the inference engine as a result of the example in this case is “multi vertices exist”.

Later on, the user needs to express the constraint:

“It is not allowed to have more than 3 end state vertices in a state transition diagram.”

Again it is assumed that the user will express this constraint using an example of three End States (Figure 4-28).

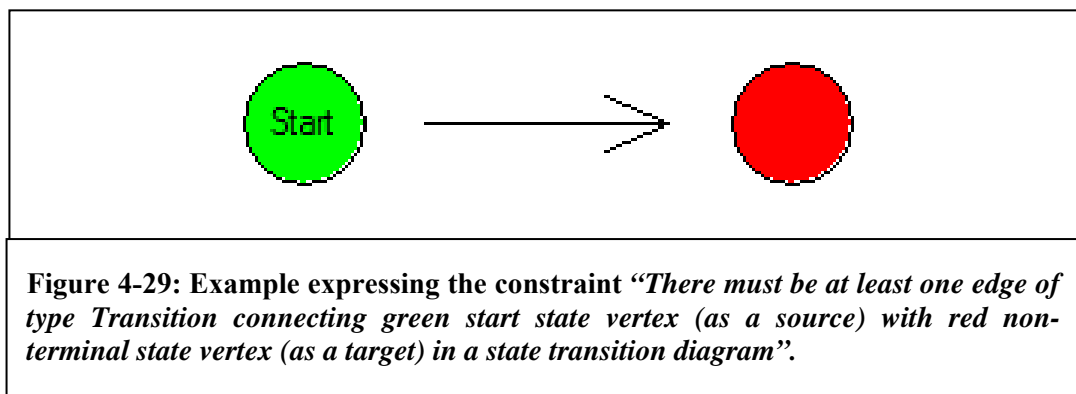


Since the same feature will be triggered in the inference engine, “multi vertices exist”, the inference manager checks the previously taught example. It discovers similarities and in this case, the generalisation process is performed on two features, the vertex type, generalised from start state to end state, and the upper bound number value is generalised from the value of 2 to the value of 3. The required constraint is inferred and generated if selected. The generalisation here is in the meaning of that the edge type and its value will be as parameters in the added rule and will be able to receive different values.

The next constraint is more complicated because it shows a problem that could not be overcome during this research. Assume the user needs to define the constraint:

“There must be at least one edge of type Transition connecting a green start state vertex (as a source) with a red non-terminal state vertex (as a target) in a state transition diagram.”

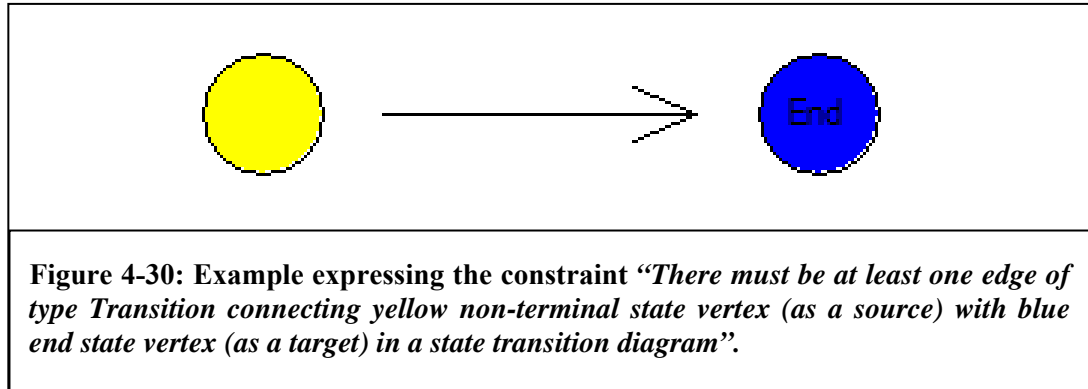
Assume that the user expresses this constraint positively using an example connecting a green start state (as a source) to a red non-terminal state (as target) (Figure 4-29).



Assume also that the user taught the system how to define this constraint because it is not implemented in its inference engine knowledge. Later on the user needed to specify the following constraint:

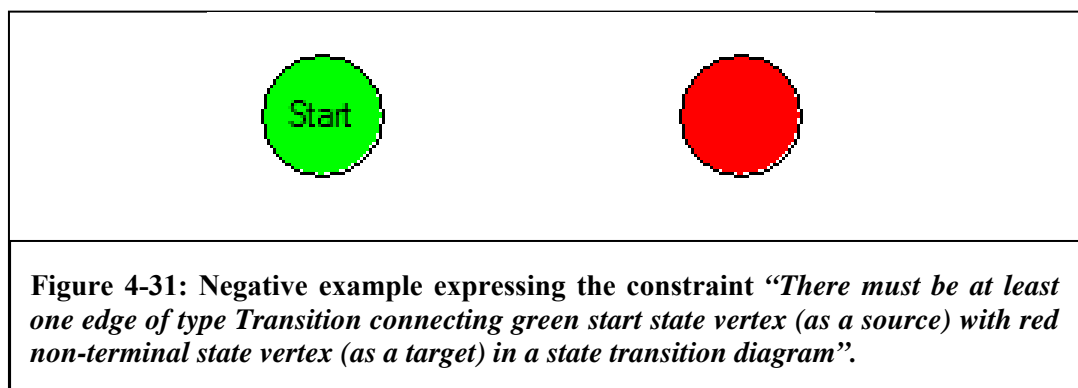
“There must be at least one edge of type Transition connecting a yellow non-terminal state vertex (as a source) with a blue end state vertex (as a target) in a state transition diagram.”

If the user uses a similar example to the one used before by connecting a yellow non-terminal state (as a source) with a blue end state (as a target) (Figure 4-30), then the inference manager will be able to detect the similarities between the two examples.

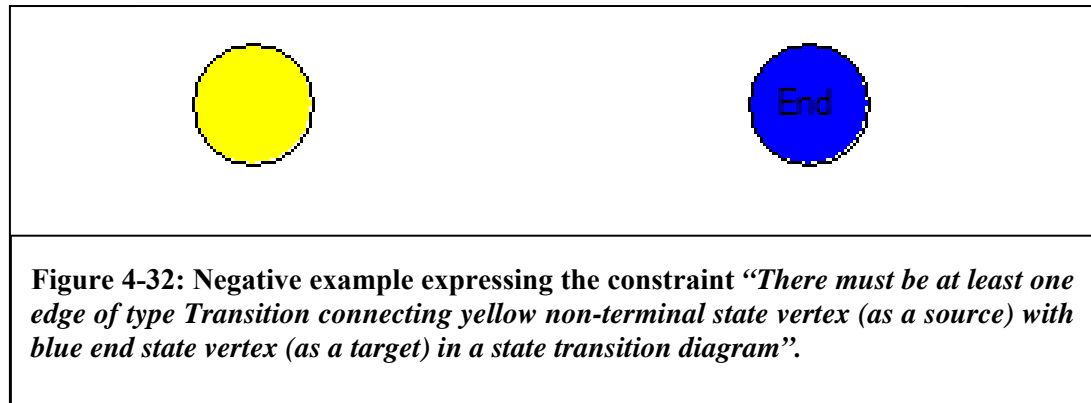


It also will be able to generalise correctly even the same vertex type, non-terminal state, has been used once as a target and in another as a source. The system will successfully generalise here for the vertex types and the colours as well.

However, what would be the case if the user teaches the system expressing the constraint negatively rather than positively? Assume that the user expresses the constraint negatively in the first place, using the example of a green start state and red non-terminal state that are not connected (Figure 4-31) and teaches the system how to specify the constraint.



Later on, the when the user introduces the second example, yellow non-terminal state and blue end state (Figure 4-32), the inference manager will not be able to recognise the source and target vertices using its current algorithm.



In this case, the system goes a step back and compares the vertex types for generalisation. Consequently, the system assumes that the non-terminal state has not been changed and no need to generalise it. The only generalisation applies to it will be the colour feature. However, the green start state will be generalised to be blue end state. The resulted constraint will be:

“There must be at least one edge of type Transition connecting blue end state vertex (as a source) with yellow non-terminal state vertex (as a target) in a state transition diagram”,

which is a wrong generalisation. This problem has no solution as the human mind even would not be able to recognise the required generalisation without knowledge in the context to be specified itself.

4.9 Conclusion

This chapter introduced the CSBE technique which depends in its core on the PBE technique. CSBE as implemented in DECS depends on an interactive, synergistic approach that encapsulates cooperation between the user and the system to specify constraints on target software model editors. This chapter introduced the general model of CSBE and its associated synergistic approach which has been implemented in DECS through a separate component, the inference manager. The synergistic approach has the advantage of keeping the user in the loop and offering alternatives in a phased manner that corresponds to the complexity of particular constraint definition tasks. CSBE technique has many distinctive features that together support the synergistic approach. These include:

- *The use of both positive and negative examples* increases the flexibility of the CSBE technique in DECS because it is more natural (or easier) to express some constraints negatively while it is more natural to express some others positively. Using both also supports the synergistic approach in a harmonic way as it allows the user to switch between interpretations. To support this feature, CSBE implements the ability to infer from the state and the action (state-action) feature which provides the power of providing more alternatives for expressing the constraints using examples of different polarities. This feature will be discussed in more details in Chapter 6.
- *Visualising the example by expressing the constraint using the target editor visual elements themselves* instead of using conceptual elements. This increases the intuitiveness and reduces the complexity of the specification and example expression processes as noted by Draheim et al. (2010).
- *Inference transparency* is another feature of the CSBE technique in DECS that allows the user to understand the way the system works through watching it working while building examples. This feature can be enabled and disabled at any time, as desired.
- DECS has a novel rule based inference engine. Rules are classified into two types, “choice” and “remodelling”. *Choice rules infer the constraints* from the introduced example and generate the constraint list that the user can choose from. *Remodelling rules are used to remodel and generalise.*
- *Visual remodelling and generalisation gives the user a visual expression of the remodelled examples which help in understanding the inference results.* The two rule types work together to support the synergistic approach by allowing the user to direct system inference.
- If the required constraint cannot be inferred, *the user can teach the system how to define new constraints* by augmenting inference engine rules or by customising the examples. This feature will be discussed in more details in Chapter 7.

In addition to the examples introduced above, DECS can infer complicated constraints. Consider the example introduced in Figure 4-33. When the user introduces an example as in Figure 4-33-a, the only reasonable system negative

inference is that the introduced structure is not allowed as in Figure 4-33-b (the inferred constraint description is not complete because it is too long). The full inferred constraint is “It is not allowed to have the structure (Start State is connected to Non Terminal State) using Transition (non Terminal State is connected to Start State) using Transition (non Terminal State is connected to Non Terminal State) using Transition (Non Terminal State is connected to End State) using Transition”.

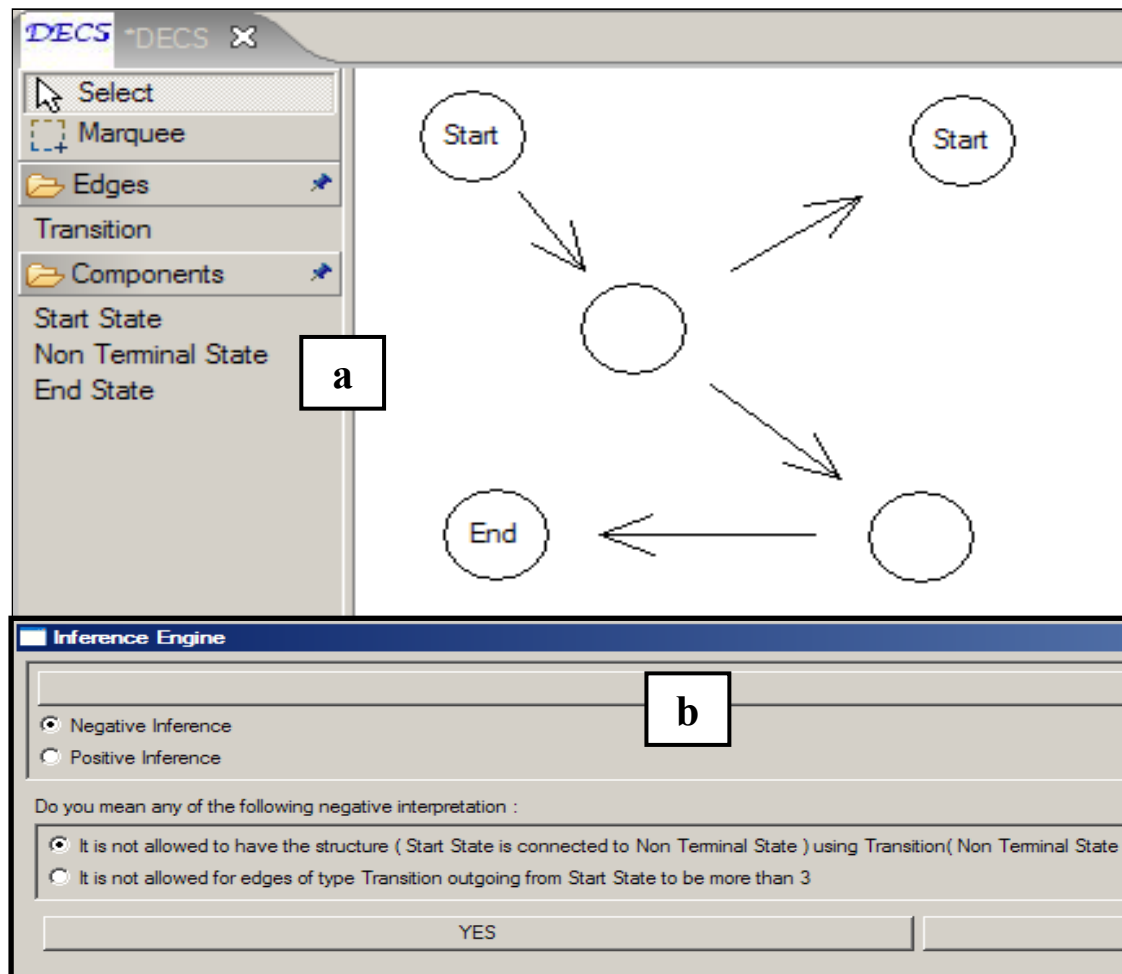


Figure 4-33: a) A complicated example in DECS. b) The inference from the example.

Finally, this research treats PBE as the core of the CSBE technique. Consequently, CSBE is a PBE technique variant with a set of features, listed above, which make it distinctive from any other PBE implementation introduced before.

Chapter 5:

Research

Methods,

Population

Definition and

General Threats to

Validity

5.1 Introduction

Throughout all the studies conducted in this research, different research methods and techniques were used for data collection purposes. These methods are summarised in this chapter. The chapter also introduces the definition of the population to which the research is generalising its results. Some generalisation issues are discussed. Finally, some general issues that threaten the validity of this research are listed and discussed.

5.2 Research Methods

The following sections document the research methods used in this research. This is considered as a documentation of the general research methods. However, study-specific research methods are discussed in each section that documents the specific study.

5.2.1 Experimental Design Principles

In all of the studies conducted in this research, a within subject design was used. This design was selected because it helps in validating the claims and hypotheses of the research. All the studies in this research involved comparing two conditions and comparing them with respect to the performance of achieving the task. A within subject design helped in:

- Eliminating the individual differences between subjects which may affect the performance as a dependent variable in all the studies.
- Measuring one of the dependent variables which is user satisfaction and estimating user preference by providing the ability to ask direct questions that compare between the two tested conditions. This is done by asking the user to answer which condition they preferred for performing the task.

5.2.2 Pilot Studies

Pilot studies were used as one of the research methods in this research. Before every study (except Study 2), some pilot studies were conducted. The number of

these pilot studies depended on the number of available subjects. However, usually three pilot studies were done. The aims of the pilot studies were different from one study to another. Estimating the required time to complete the task, ensuring that the users understand the task as it is written on the task sheets and from explanation written on a whiteboard by the researcher, estimating the learning time, and ensuring that the task in general is feasible are the common reasons why pilot studies were used in this research.

One pilot study (in Study Three) aimed also to discover the suitable conditions that should be used in the experiments. In this study there was a need to make a decision regarding one of the conditions to compare with. This condition had two alternatives and the pilot study helped in providing more data that allowed the experimenter to take the decision and support it. This gives the pilot studies a major contribution in the design of the experiment.

The pilot studies were of benefit sometimes in a way that was not expected. For example, in some of the pilot studies (in Study Four) the subjects interacted with the system in a way that indicated confusion. After investigation, it was found that the subjects had done a previous experiment in the same research (Study Three) and they were expecting something from the system but the system did not react in the way they expected. Accordingly, the study was very confusing to them as they wasted a lot of time just repeating the “examples” trying to find out the difference in the system behaviour between last time they used the system and this time. Based on this pilot study, it was decided that all the participants of the study (Study Four) should not have experienced DECS or conducted any experiment in this research before. In this case, pilot studies helped to avoid a problem in the experiment and to eliminate a threat to validity.

5.2.3 Screen capturing and recording

Purpose: collecting effectiveness and efficiency data.

This technique was used instead of logs to collect the required data. The users' screens were captured as a video which recorded their work to achieve the task they were asked to perform. Camtasia Studio® software was used for the purpose of screen recording. Every user was given the task and the screen recording software

was started. When the user finished the task, the recorder was stopped. Later on, the required data was collected from the recorded videos. Using a screen recording method for data collection had many advantages that can be summarised as follows:

- Since the tasks were different from one study to another it was easier to record the task and extract the required data later on than implementing a log for every study.
- Eliminating the dangers of implementation errors and bugs in building the log. Instead, recording the videos allowed the researcher to guarantee that the required data will be recorded and will be independent from any implementation.
- It was easier to measure the correctness using the video than the log, especially when using the form-filling technique. When using the video the correctness of the constraint specification was decided directly instead of revising log files and searching for the errors.
- In measuring the required time, it was more convenient to depend on videos recorded to decide where to start counting the time required to achieve the task and when to stop counting.
- Using the recorded screens allowed measurement of the correctness and the time when the users were given a limited time to finish the task. Some users used more than the time limit trying to complete all the constraints. This is because sometimes the users found the tasks as an interesting challenge. Conducting the measurements using log files required many comparisons to check the limit and the time last constraint was defined for each user. Such complications were avoided using the screen recorder.
- It eliminated the dangers of missing logging some required data. This sometimes happened because of an implementation bug or something that does not come to the mind of the designer before the experiment. Later there would be a need to repeat the experiment to be able to log the required data. In the case of the screen recording method, if more data is needed, the researcher may watch the videos again to extract the required data.
- It allowed some mistakes in collecting data to be avoided by discovering implementation bugs in the system. This happened only once (in Study 3) when a bug in DECS was discovered while watching the videos at data collection time.

Screen recording helped in this case also by repeatedly watching all the videos to determine the effect and spread of the bug in the experiment. This helped in taking the decision of eliminating the data of one of the constraints in Study 3.

- The videos recorded all the interaction behaviour of the user with the system during the task which may open an opportunity for additional analysis of this data from another perspective. This may include analysing the data on the constraint level to discover the differences between the constraints. The data can also be analysed from the human computer interaction perspective to study the interaction with CSBE's user interface.

5.2.4 Questionnaires

Purpose: user satisfaction data collection

In addition to the quantitative data collected using the screen recording, there was a need to measure user satisfaction as one of the measurements required in this research.

Post-Experiment Questionnaires: In three of the four studies conducted throughout this research (all the studies except Study Two), questionnaires were used to evaluate user satisfaction about the technique or condition used independently from the other condition. This is done by asking the users to fill a questionnaire after each condition used. Because the conditions were counterbalanced, this is considered as a measurement of the user satisfaction with each condition alone. The questionnaires depended on Likert scale of 5 questions. This type of questions was selected to elicit the degree of user agreement or disagreement with the statement (or question) in the questionnaire. It was also a suitable way of measuring the attitude of the user regarding the point they are asked about.

Exit Questionnaires: In addition to the questionnaires after each condition, users were asked to answer one more questionnaire after finishing the whole study (both conditions). Most of the questions in this questionnaire asked about the preference of the user to one of the conditions tested in the tasks. This helped in measuring user satisfaction and preference to one of the conditions. One study (Study Two) used only the last type of questionnaire because it is comparing comparison

between two constraint expressions in natural language and its aim was to compare between the two expressions with respect to user preference. These questionnaires also were composed of Likert 5-scale questions to collect the attitude of the user towards any of the independent tested conditions.

Post-Constraint Questionnaire: In one of the experiments, a short questionnaire was used with the purpose of collecting data about the effect of constraint expression in natural language on expressing the constraint using CSBE. This was the only questionnaire used for the purpose of collecting data about user satisfaction. Again, this type of questionnaire depended on Likert 5-scale questions.

5.2.5 Interview

Purpose: user satisfaction data collection.

Semi-structured interviews were conducted with the users after each finished filling in the exit questionnaire. This interview was included in three of the four studies conducted in this research. It was used as a qualitative method to extract and collect data regarding user satisfaction. In the interview, the users were asked to give their opinion of the system in general and in the conditions they experienced. The interview was valuable in collecting data about user satisfaction outside the limited questions prepared and designed in the questionnaires. Some of the opinions were included as part of the discussions of the experiments to support the research claims and to provide answers for the research questions.

Although the interview was semi-structured and while the user was interviewed, new questions and opinions may be asked to document; there were four questions that were prepared for the user to answer if no independent opinions were given. These questions were “What did you like about the system?”, “What did you dislike about the system?”, “If you asked to change one thing in the system, what would that thing be?”, and “Do you have any other comments?”. The questions were general enough to open discussions and other questions about the system. These questions are considered as open questions; however, they were part of the interview. This type of question and interview were selected because they give the user the freedom to express and document their feelings and comments about the system. In addition it gave them the ability to give opinions and feedback about things that might

not be covered in the Likert scale questions. This type of questions also gave another way of system evaluation than limiting the subject with a number to select based on a specific question.

5.3 Sample Selection

5.3.1 Population Definition

In this research the population which the samples were selected from was well-defined. The population was defined by the following characteristics:

- Participants should have knowledge of diagram editor CASE tools and their use.
- Participants should have practiced software modelling before and they should know how to use at least one of the available (commercially or free) diagram editor CASE tools.
- Participants should have knowledge of the diagrams used in the studies (viz., State Transition Diagram and Use Case Diagram).
- Participants should be able to speak, read and write English.
- Participants were NOT a required to have experience or knowledge in constraint definition or its domain such as knowledge of a constraint specification language.

The above forms the specification of the population for this research. However, some extra conditions were introduced for specific purposes of some studies that cannot be considered as part of the general definition of the population. An example of this is the requirement in (Study Four) for subjects that have not been used DECS before or done an experiment related to it through this research. Such specific requirements are documented in each study. Apart from such study-specific conditions, it is believed that the above criteria characterise and define properly the population that this research depended on. Jorgensen & Sjoberg (2004) document that it is not possible to make inferences and generalisation from studies results which do not have a well-defined population. They also state that “empirical studies of software engineering may seldom have well-defined populations.” They suggest that the difficulty of defining the population arises from unclear elements that should be considered in the population definition.

Again, it is believed that the above criteria are sufficient to define the population from which the samples for this research should be drawn apart from the specific criteria that should be available in some studies. Some other elements that may be important in some other studies were not included because they are considered of insignificant importance in this research. These elements may include the mother language, the gender, the age and the health condition.

5.3.2 Deviation from the Well-Defined Population

Based on the description, it is believed that this research has a well-defined population from which random samples should have been drawn. It is believed also that the samples selected to conduct the experiments of this research are drawn from this well defined population. However, all the participants were postgraduate students, either taught course or research students, following programmes in Computing Science or Software Engineering. This may be considered a deviation from the well-defined population by sampling from only one subset of it instead of depending also on professionals as participants. This is because the researcher has only access to this subset and did not have access to professionals. This deviation from the well-defined population may affect, according to (Jorgensen & Sjoberg, 2004), the ability to generalise and infer from testing the statistical hypothesis.

However, all the participants satisfied the characteristics that define the target population. Based on this, it is believed that the selected sample represents the required population because the tasks in the studies do not depend on knowledge or skills that are different between students as representative of inexperienced professionals, as (Sjoberg, et al., 2005) described them, and experienced professionals. However, the participants' attitude towards CSBE may be different if they are experienced professionals who have already invested considerable effort in learning a constraint language, for example. According to this, this research used samples that represent the well-defined population which allows generalising its results. In a survey of 103 published empirical studies, (Sjoberg, et al., 2005) surveyed the documentation of 20 replications of 14 experiments. Among these, three experiments originally used students as subjects and, when replicated, used professionals as subjects. In the three replications, the results of the original studies were confirmed. This provides some empirical evidence that the community of

students can in some cases be representative of a wider software engineering population.

5.3.3 Drawing Samples from the Population

Samples were drawn from postgraduate students and researchers in Computing Science and Software Engineering. They were considered as representative of the population. The technique of sampling was completely random using email. For each study (except Study Two), an email was sent to the postgraduate taught, research students and researchers (not students but none of them participated). All the requirements of the subjects, such as software engineering background, knowledge of CASE tools and knowledge of modelling diagrams, were included in the email. The email also contained the information sheet as attachment. Each participant was paid £10 at the completion of the experiment and the consent form indicated that the payment would be made even if the subject chose to withdraw from the study before completion. However, this did not happen with any participant. The first 16 students that replied to the email and satisfied the definition of the required population were taken as subjects. This procedure was followed in all the studies except Study Two. The number of participants (16) for each study (except Study Two 37 participated) was selected because of the available resources of time in waiting for collecting the participants, time in conducting the studies, analysing the data and the available fund for paying the participants.

In Study Two, which is an online study that used a questionnaire, there was a need to specify the target population or the well-defined population. For this study, the population is the same as the rest of the research. To be able to direct the questionnaire to the required population, the same technique used for other studies (the email) was also followed in this study. In Study Two, an email was sent to all the Software Engineering and Computing Science students in addition to the researchers in the School of Computing Science at Glasgow University. This was to ensure that the participants had knowledge of State Transition Diagrams. The email contained a hyperlink that directs to the online questionnaire (the experiment). To increase the sample size, the researcher also sent the email to some colleagues who are studying research degrees in the UK (Bradford University) and asked them to broadcast the email to the Computing Science research students in their institution.

5.4 Threats to Validity

In addition to the research methods that were used generally in almost all the experiments, there were some general threats to validity that apply to almost all the studies. Similarly to the specific research methods for each study, threats to validity relevant to a particular study are introduced at the end of the description of that study. The following list describes the general threats to validity of this research.

- ***Using students only:*** The research samples contained students only instead of containing professionals and students. This may threaten the validity of the research by affecting the statistical test results generalisation. Sjoberg, et al. (2005) claimed that students may be considered as representative of junior professionals with no experience. This can be the case in this research because the students that participated as subjects were primarily postgraduate students with three or four years of university-level education. Only one final year undergraduate student participated in one of the studies (Study One). Additionally, some of these student participants had worked at companies before they undertook their postgraduate studies; however, only the current state of the subjects were taken into consideration and information about previous work experience was only by personal communication with the research after the experiment. Part of Study One was conducted in Jordan, also involving postgraduate students; however, these participants were also working at the same time at a university (the Jordanian Applied Science University) as lab supervisors. This means that all the participants (except one) were involved in postgraduate studies which indicates they can be considered as professionals, albeit with no known practical software engineering experience. Based on this, generalising the results of the studies is threatened by the fact that all the participants were students. Sjoberg, et al. (2005) introduce that involving low proportion of professionals software engineers in the studies reduces their realism. However, the nature of the population definition for these studies, discussed in section 5.3, would suggest that any threats to validity lie in generalisation of preferences rather than task performance.
- ***Relativity of subjects with the researcher:*** All the participants, except those conducted the experiment in Jordan and part of the participants in Study Two,

were studying at the same university as the research running the studies in (Glasgow University). Some participants in one of the studies (Study Two) also included friends of the researcher. This may threaten the validity of the results because of the relation between the researcher and the participants. In an attempt to limit this threat, only one study was conducted every year and most of the participants were from the newly enrolled masters students. These students had met the researcher for the first time at the time of the experiment and they had no relation with him before. To limit the effect regarding the research students at the same department as the researcher, they were not told nor had any hint about the expected results or the preferred independent condition by the researcher. The participants from Jordan were also met for the first time at the time of the experiment and had no relation with the research before. Some of the participants in Study Two were friends with the aim of increasing the sample size. The precautions of not telling them anything about the expected or the preferred results were taken.

- ***Bias towards the implementation:*** According to an expert (Prof. Sjoberg, D.), most of the empirical studies that involve tools developed (implemented) by the researchers themselves, the results are always to the advantage of their implementations. Sjoberg, et al. (2005) document 20 replicated empirical software engineering experiments; 11 were conducted by the original authors and 9 by others. All the 11 experiments replicated by the original authors confirmed the original results. However, between the 9 experiments replicated by others, 6 reported different results from those obtained in the original experiment and one reported partially different results while only 2 experiments confirmed the original results. This suggests there may be a bias towards the CSBE technique and its features over the form-filling technique and other tested features in this research. While this potential bias may well exist, form-filling interaction techniques follow a set of conventions, largely imposed by the interaction components supplied by the user interface libraries. This set of constraints reduced the degree of freedom of the researcher in the design of the form-filling condition. In general, the form-filling technique is a typical representative of its type.

5.5 Conclusion

This chapter introduced the research methods that are used in almost all the studies in this research. These methods include pilot studies, screen recording, questionnaires and interviews. These methods were discussed with clarification of the rationale for the use of each of them with reference to the aims of this research. The chapter also introduced the definition of the participant population by listing the defining characteristics of this population. The chapter introduced the sampling technique used in this research and discussed the deviation from the well-defined population because the samples contained only a subset of students, indicating the aspects of generalisation that are potentially affected by this limitation. Finally the chapter discussed the general threats to validity including using students as subjects, the relationship of participants to the researcher and a potential “implementation” bias introduced because the researcher produced the representative control condition in one of the experiments.

Chapter 6

Empirical Studies

6.1 CSBE vs Form-Filling Technique (Study One)⁹

6.1.1 Introduction

This section presents an initial study to investigate the usefulness of Constraint Specification by Example (CSBE) for constraint definition. This is achieved through an empirical study that compares CSBE with a typical form-filling specification technique represented as a wizard and tabbed forms. In addition to the justifications introduced in Section 3.5.1, the form-filling technique was chosen to compare with because it provides more support for the user than free text-based techniques in constraint specification which require the user to learn a textual language to be able to express the constraints. In other words, it is a good and feasible technique to compare with. By contrast, comparing with a free text-based approach would provide weaker evidence of the performance of CSBE.

Study One was designed to answer the first research question:

Does CSBE improve the performance of constraint specification in a meta-CASE tool compared to the form-filling technique?

Answering the question achieves one objective of this research which is to study the performance in terms of effectiveness, efficiency and user satisfaction of the CSBE technique in comparison with a typical form-filling constraint specification technique. The empirical study tests the claim that *the performance of specifying constraints in a meta-CASE tool using the CSBE technique is better (higher performance) than using the form-filling technique.*

As previously discussed, the form-filling (represented by a wizard and tabbed forms) and CSBE techniques were implemented in DECS. This section details the

⁹ The work discussed in this section has been published as Qattous, Gray and Welland, 2010

empirical study that compares both techniques and shows the results. At the end it discusses and concludes from the results with some comparisons with related work.

6.1.2 Experimental Methodology

6.1.2.1 Aim and Hypothesis

The aim of this research is to reduce the complexity, and to facilitate and simplify the constraint specification task within the context of meta-CASE tools using Constraint Specification by Example (CSBE). For this purpose, an experiment was conducted to evaluate the constraint specification performance of the novel technique, CSBE, in comparison to the common, form filling, technique (using a ‘wizard’ and ‘tabbed forms’). The following potential dependent variable measurements were tested for each technique (the CSBE and form-filling) to conduct the performance evaluation.

- The *effectiveness* in terms of the resulting constraint specification *correctness*.
- The *efficiency* in terms of the *time required* for accomplishing the constraint specification task.
- The *user satisfaction* with the technique.

6.1.2.2 The Hypothesis

The null hypothesis of this experiment states that,

H0: *there is no difference between the techniques regarding the variables to be measured.*

The alternative hypothesis states that,

H1: *CSBE performs better than the form-filling technique with respect to all variables measured,*

6.1.2.3 Collection and Tasks

The study required users to carry out a set of supplied constraint definition tasks. Two diagram types were selected as target visual languages to be specified: State Transition Diagram (hereafter STD) and Use Case Diagram (hereafter UCD).

The experiment was conducted on both diagram types separately and at different times. The experiment on STD was conducted in Scotland while the UCD experiment was conducted in Jordan and in Scotland and, consequently, had more participants. Both diagram types, STD and UCD, were selected because they are commonly used in the software design process and all the participants were familiar with them. The diagram types also contain all the general types of constraints that may appear in most other diagram types.

A main constraint list was created for each diagram (Appendix B). Each main list contained constraints that define an entire diagramming language. Some other constraints were added to each list as customised constraints that define potential customised. Constraints in each list were divided into six different groups based on similarities between constraints. The constraint groups were organised based on the following criteria:

- Constraints related to cardinality of vertices. This class contains constraints that define the upper bound and lower bound numbers of allowed vertices such as the constraint *“It is not allowed to have more than one Start State in the diagram”*.
- Cardinality of incoming and outgoing edges and connections between vertices such as the constraint *“The End State must only have incoming transition edges”* and *“It is not allowed to connect two vertices of type Actor using an Association edge type”*.
- Unique visual representation of vertices such as *“The End State must have a unique visual representation”*.
- Label-related constraints such as *“Each Non-Terminal State must have a label”*.
- Path-related constraints such as *“There should be a path between the Start State and every other vertex in the diagram”*.

Some of these categories generated more than one group, such as the second group which is divided in the UCD into two groups (outgoing and incoming edges) with a logical operator. One criterion, unique visual representation, forms a group by itself and was used only in STD. The full constraint lists are given in Appendix C.

In order to create a set of constraints for use in the experiment, one constraint was picked randomly from each group to form a list of six constraints. The chosen constraint was removed from its group to ensure participation of all the constraints in the lists. When a group had no more constraints, all the removed constraints were returned back to start choosing again. Of course, each group has a different assigned number of constraints to choose from. Using this method of grouping and constraint selection, it was guaranteed that all the users will get similar, but not identical, lists of constraints. This reduces the differences between the constraint lists with different users and used a wide range of constraints for each diagram type. The resulting lists of 6 constraints each were used in the experiment as the basis of the user tasks.

6.1.2.4 Participants

41 participants (16 used STD and 25 used UCD) were selected from Computing Science and Software Engineering graduates and postgraduate students in Scotland and in Jordan. In the Scottish study, one final year undergraduate student also participated. The difference in number of participants for each diagram refers to the use of nine extra participants on UCD for more data validation.

6.1.2.5 Experimental Design

For this evaluation a within subject design was adopted. The constraint lists were randomly assigned to different participants. For each participant, two different constraint lists were used, one for the training and the other for the constraint definition task. The users' task was to define the 6 constraints in the list provided. The same list was assigned to each subject and used for both conditions, the form-filling and the CSBE, to eliminate any effect of single constraints on the technique evaluation.

The order of the technique usage was counterbalanced in an attempt to limit the order effect associated with the techniques. Each participant was trained for about 20 minutes on each technique and allowed to carry out the training task without assistance before starting the main task. The training constraint list was used for this purpose. The same process was repeated for each technique. The participants were asked to fill out a number of different questionnaires at different stages of the experiment. A time limit of 15 minutes was imposed for each task. Each evaluation

was recorded by screen capture to be used later for data extraction and analysis. The experimenter also observed each session and took notes.

It should be noted that the experimenter intervened on several occasions when participants exhibited signs that they could not progress with the tasks. This only occurred in the wizard condition and help was solely in the form of hints about the labels of different properties in the wizard. No further interference or help was provided.

6.1.3 Results

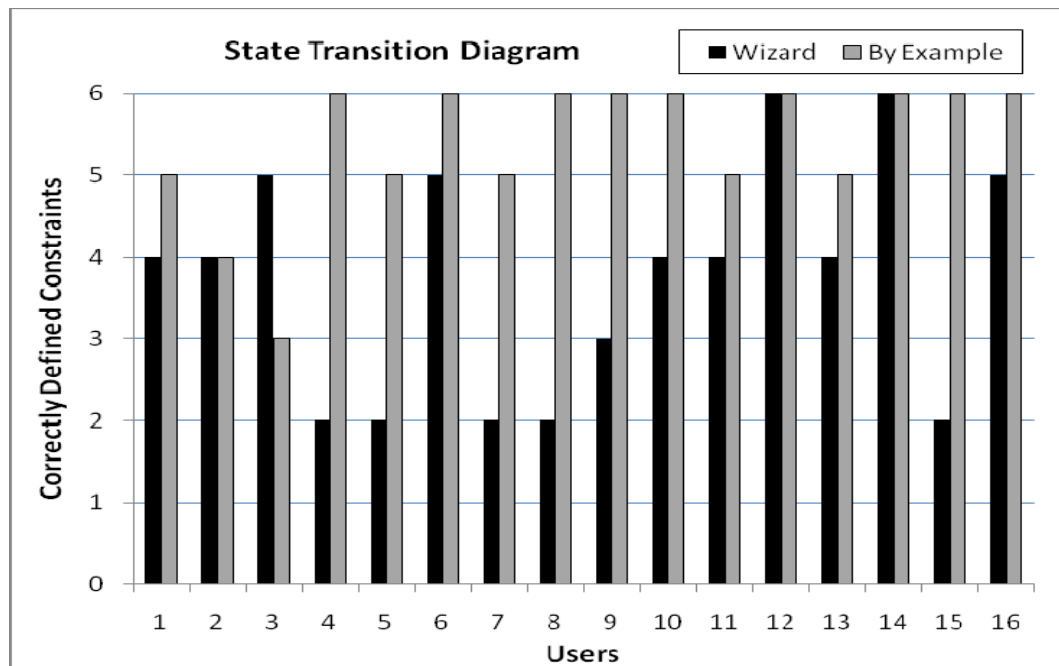
All the 41 participants were familiar with the diagram types used in the evaluation. Most of the participants indicated that they were familiar to some degree with the constraint definition task, with an average mean of 3.0 and 4.0 on a scale of 5 for STD and UCD respectively. Each experiment took approximately 2 hours including the training time. The results for the users' attempts were analysed with respect to the above mentioned hypothesis. The nonparametric Wilcoxon signed-rank test was used to analyse correctness and satisfaction results while a survival analysis with the log-rank test was used to analyse the time results. The comparison between the two techniques regarding the required measurements is presented in the following sections.

6.1.3.1 Correctness

The number of correctly defined constraints for each participant in both diagram types was gathered from the recorded screen capture videos. The constraints that were not attempted by a participant because of the time limit (the 15 minutes given to accomplish the task) were considered as defined wrongly. Apart from this time limit problem, all the participants attempted all the constraints. Figure 6-1 shows the number of correctly defined constraints for each user in the STD and Figure 6-2 shows the same results for the UCD. Both figures show the results for the wizard and the CSBE techniques. A significantly higher number of constraints were specified correctly using the CSBE than via the wizard.

6.1.3.1.1 State Transition Diagram

In the case of the STD, 12 out of 16 (75%) of the participants defined a higher number of constraints correctly using the CSBE technique than the wizard. Nine users defined all the constraints correctly in the task using the CSBE technique, two of whom also did using the wizard (Figure 6-1).



	Wizard	CSBE
Mean	3.8	5.4
Min	2.0	3.0
Max	6.0	6.0
STD	1.4	0.9

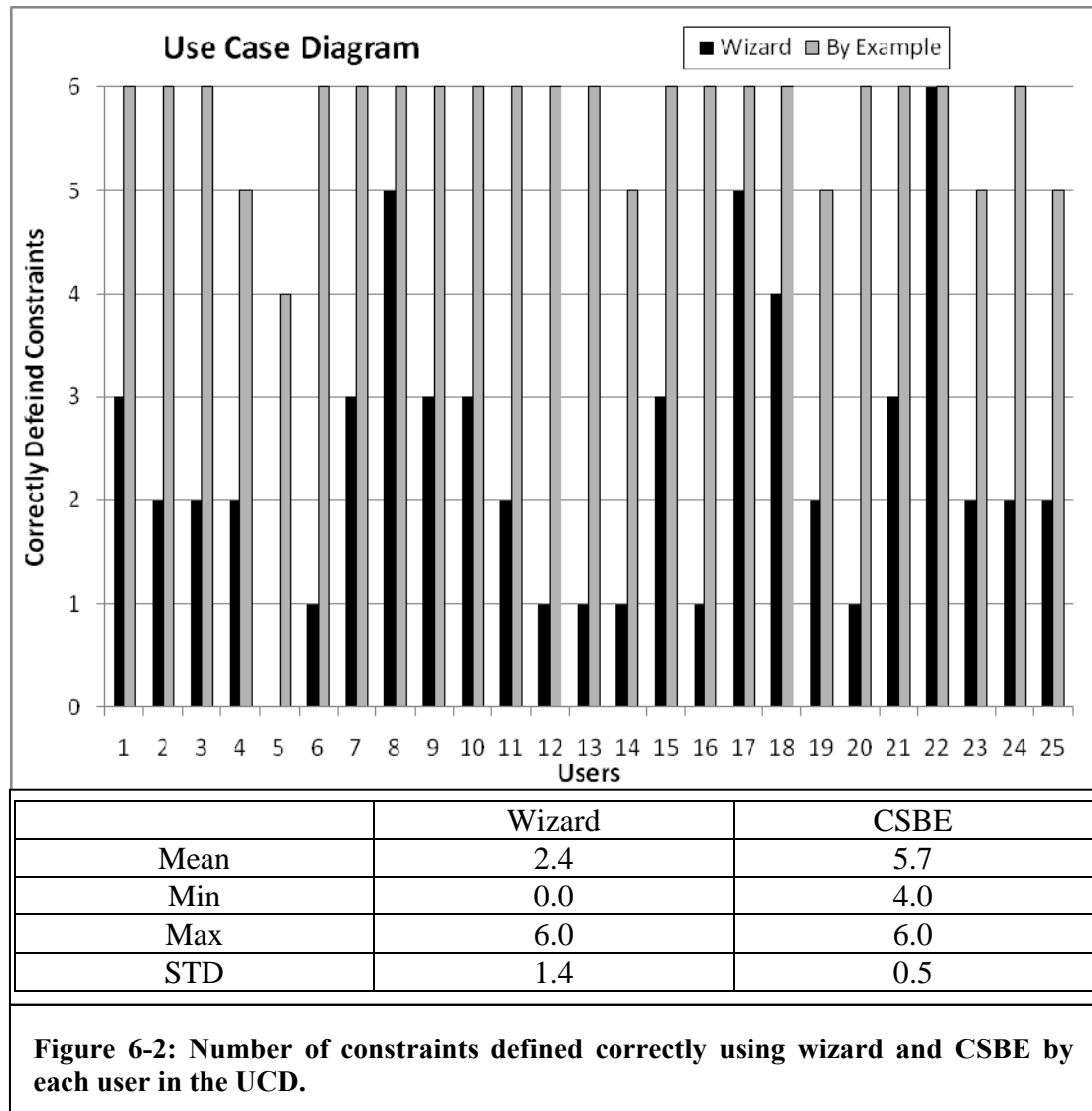
Figure 6-1: Number of constraints defined correctly using wizard and CSBE by each user in the STD.

The average percentage of constraints defined correctly using the wizard technique was 62.5% (3.8 constraints) while it was 89.6% (5.4 constraints) using CSBE. Analysis shows that there is a high significant difference between both techniques regarding correctness in constraint definition ($p < 0.01$).

6.1.3.1.2 Use Case Diagram

In the case of UCD, 24 users (96%) defined a higher number of constraints correctly using the CSBE technique than the wizard technique. Only one user defined

the same number of constraints correctly using both techniques. 19 users defined all the constraints correctly in the task using the CSBE technique, and one only did using the wizard (Figure 6-2).



In the case of the wizard, the average percentage of constraints defined correctly was 40% (2.4 constraints out of 6) while it was 95% (5.7 constraints out of 6) using CSBE. Analysis shows that there is a high significant difference between both techniques regarding the correctness in constraint definition ($p < 0.001$).

6.1.3.2 Time

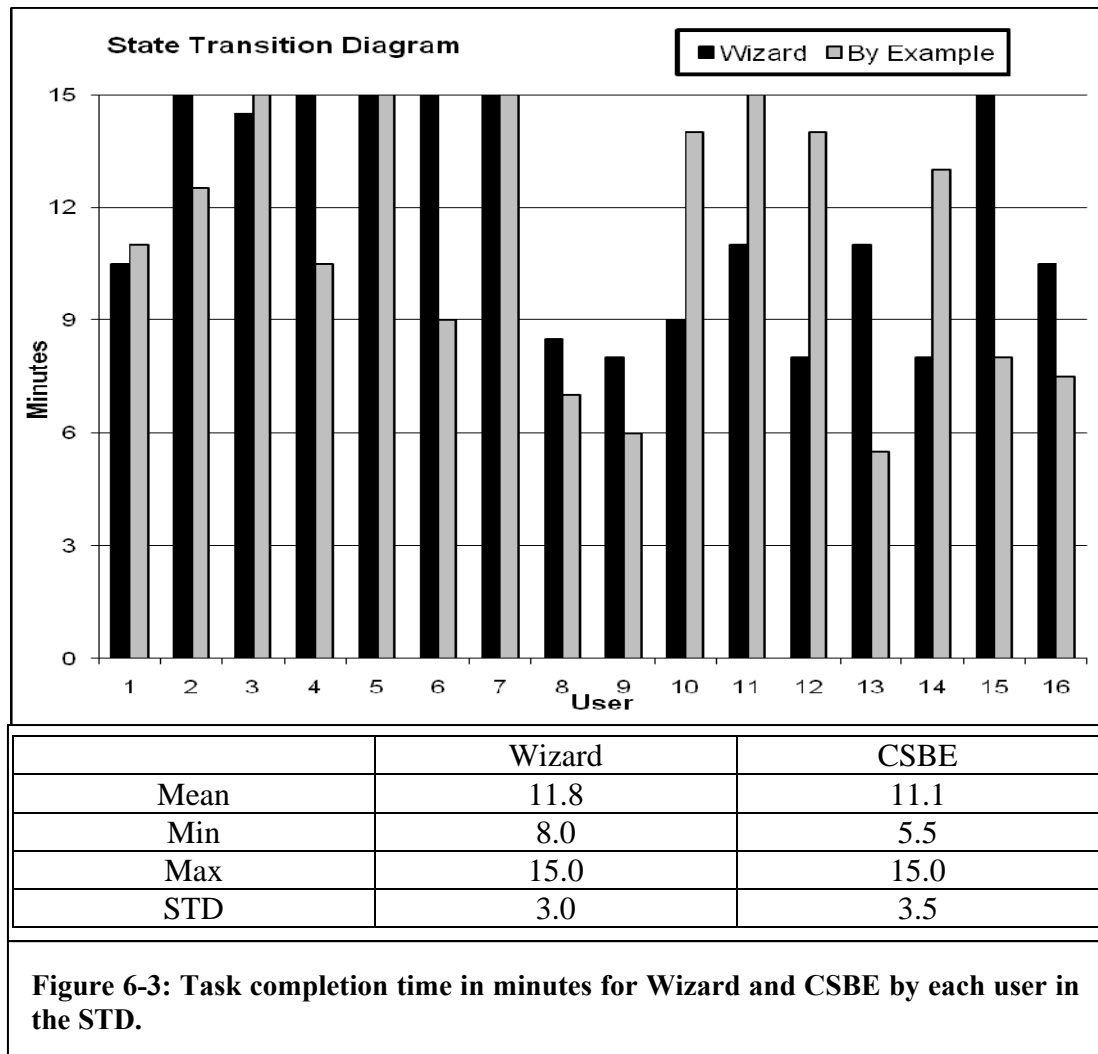
The time required to accomplish the constraint definition task for each participant in both diagram types was gathered from the recorded screen videos. The

time for each participant was rounded by a factor of 15 seconds. This means if a participant accomplished the task using 10 minutes and 14 seconds, that was rounded to 10 minutes but if the time used was 10 minutes and 15 seconds, this was rounded to 10.5 minutes. Figure 6-3 shows the time required for each user to accomplish the task in the STD and Figure 6-5 shows the same results for the UCD. Both figures show the time for the wizard and the CSBE techniques. In both diagram types, the CSBE produced better results (higher correctness, less time required, higher user satisfaction) than the wizard.

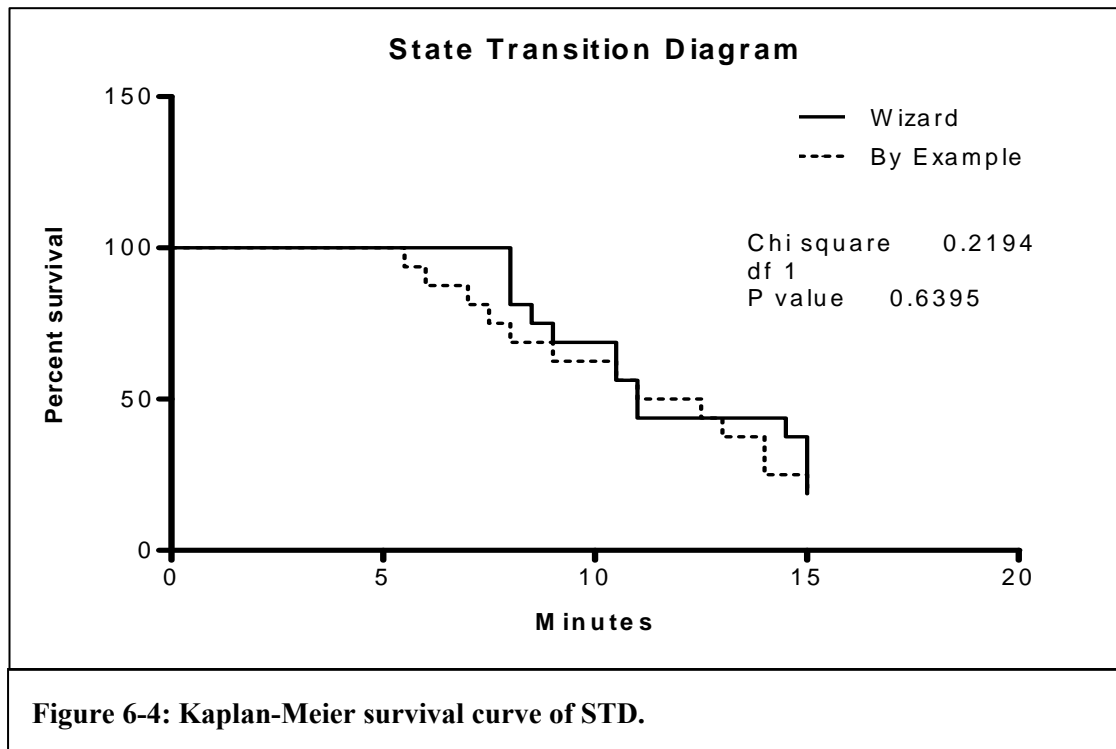
As mentioned above, 15 minutes were given for each participant to accomplish the task. However, some of the users did not finish the task within the 15 minutes and some others required the full time and finished exactly after 15 minutes. For these users, the statistical analysis, survival analysis, was required to measure the significance of difference between the both techniques regarding the time required to accomplish the task. If such data had been ignored or excluded from the data analysis, this would have introduced a selection bias to the experiment. In survival analysis, the data for participants that did not finish their tasks or subjects that stopped doing the experiment before the measured event has happened to them is called right-censored data. Survival analysis using the Kaplan-Meier survival curves, which show the percentage of the population still surviving at a giving time point, was conducted on the time data for STD and UCD and presented in Figure 6-4 and Figure 6-6 respectively. The Log-rank test was used to calculate the significant difference between the two Kaplan-Meier survival curves for both techniques.

6.1.3.2.1 State Transition Diagram

In the case of the STD, 50.0% (8 of 16) of the participants accomplished the task in less time using the CSBE technique while 37.5% (6 of 16) accomplished the task in less time using the wizard. Three users required the 15 minutes maximum time and other three required more than 15 min to accomplish the task in the wizard. Only 1 required the 15 minutes and 3 required more in the case of the CSBE (Figure 6-3).



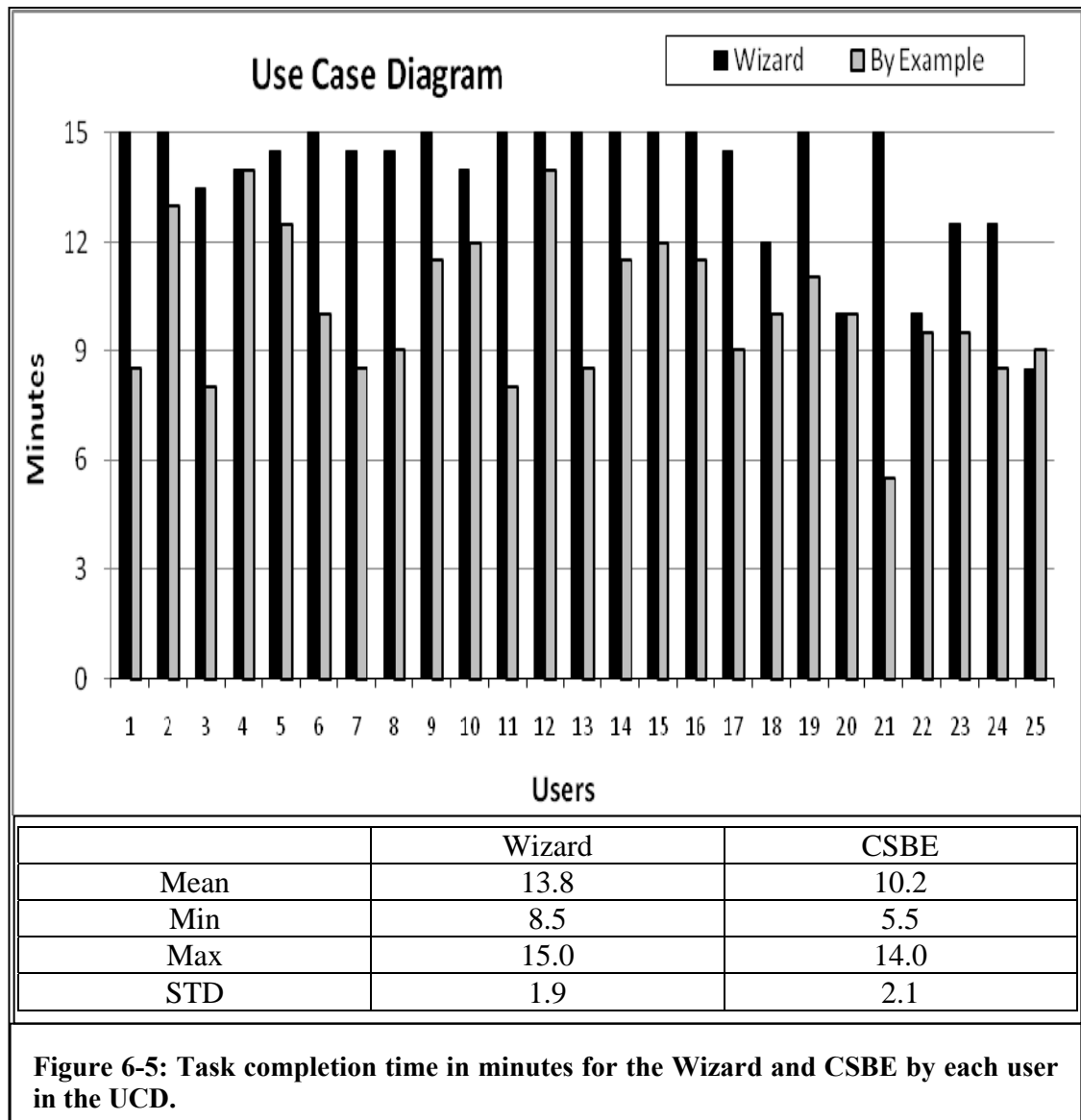
Two users required the 15 minutes allowed or more in both techniques and they were the only users who required the same time for both techniques. One of them finished the task using the maximum 15 minutes in CSBE but not with the wizard while the other did not finish the task in either case.



The Kaplan-Meier survival curve for STD (Figure 6-4) shows that both curves are close to each other and have almost the same trend. However, it also shows clearly that CSBE curve is under the wizard almost all the time which indicates that a higher number of users finished their tasks using CSBE in less time as also shown in Figure 6-4. The p value (= 0.64) shows that there is no significant difference between both techniques regarding the time required to accomplish the task.

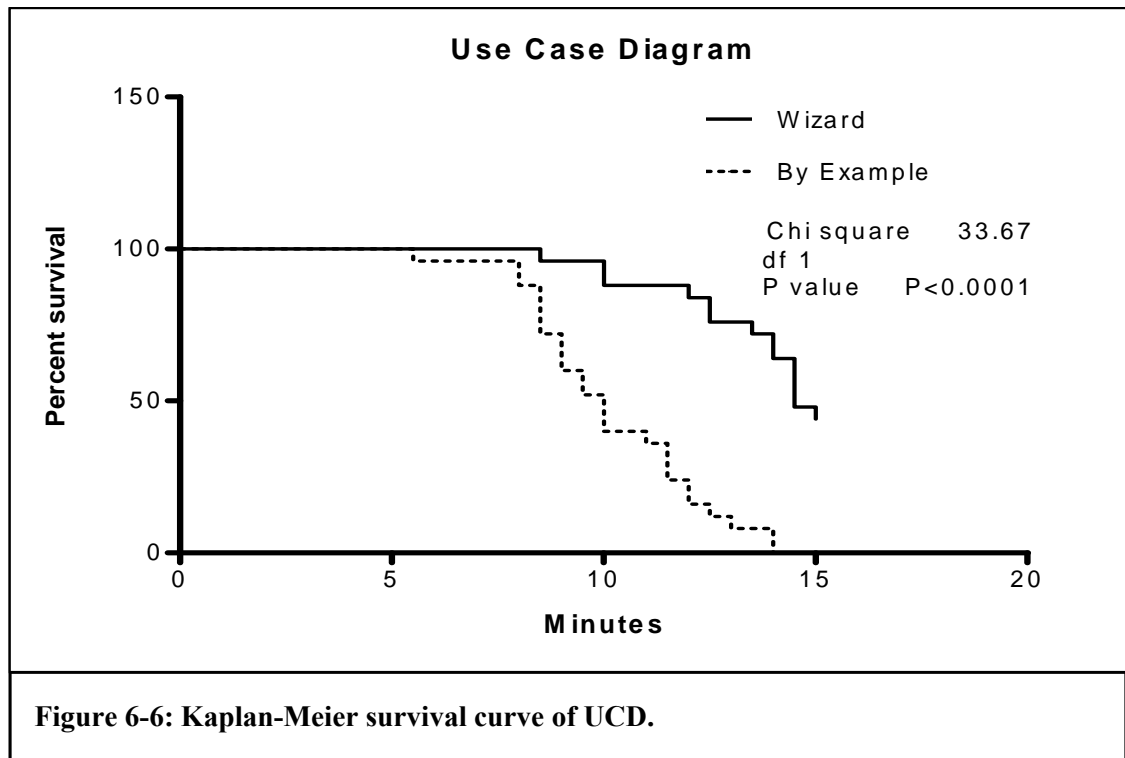
6.1.3.2.2 Use Case Diagram

In the case of the UCD, 88% (22 of 25) of participants accomplished the task in less time using CSBE while two participants required the same time, 14 and 10 minutes, in both techniques.



11 users used more than the 15 minutes given and one required exactly 15 minutes using the wizard technique; while all the participants finished before consuming the given time in the case of CSBE. The results suggest that participants have performed more quickly with the CSBE technique than the wizard (Figure 6-5).

The Kaplan-Meier survival curve for UCD, (Figure 6-6), shows a distinct difference between both technique curves. CSBE survival curve is always under the wizard one which indicates that the number of participants who accomplished the task using CSBE is higher.



The figure also shows that none of the participants failed to accomplish the task within the required time using CSBE technique. The p value (<0.0001) shows that there is a highly significant difference between both techniques in the time required to accomplish the task.

6.1.3.3 User Perception

In order to provide further validation for the hypothesis, post task and post experiment questionnaires that the participants filled out were analysed.

Table 6-1: User perception of wizard and CSBE techniques in STD and UCD. (higher = better (higher user satisfaction), bold and underline = significant difference)

	State Transition Diagram		Use Case Diagram	
Questions	Wz	CSBE	Wz	CSBE
How successful were you in accomplishing what you were asked to do?	3.7	<u>4.3</u>	2.5	<u>4.8</u>
This technique was powerful enough to allow me to define my constraints.	3.9	<u>4.3</u>	4.3	<u>4.7</u>
In most cases, I was confident that I defined the required constraint.	3.3	<u>4.6</u>	2.4	<u>4.8</u>
I felt that I acquired experience in this technique quickly while I was working.	4	4	3.6	<u>5</u>
I am satisfied with my performance in constraint definition tasks using this technique.	3.5	<u>4.1</u>	2.4	<u>4.8</u>

Table 6-2: User perception of wizard and CSBE techniques in STD and UCD. (lower = better (higher user satisfaction), bold and underline = significant difference)

	State Transition Diagram		Use Case Diagram	
Questions	Wz	CSBE	Wz	CSBE
How mentally demanding was the task using this technique?	3.5	<u>3.1</u>	4.2	<u>1.4</u>
How hurried or rushed was the pace of the task using this technique?	2.8	<u>2.3</u>	3	<u>1.2</u>
How hard did you have to work to accomplish your level of performance?	3.1	<u>2.4</u>	3.9	<u>1.4</u>
How uncertain, discouraged, irritated, stressed, and annoyed were you?	2.6	<u>1.8</u>	3.8	<u>1.2</u>
While I was working, I felt that I needed help from an expert.	2.9	<u>2.1</u>	4.3	<u>1.3</u>

In the post task questionnaires, participants' opinions about the constraint definition task using each technique were investigated. There was a need to discover the feelings of the participants while interacting with the system in the case of each technique. A Likert 5-point scale was used and some of these were inverted to reduce bias. Table 6-1 and Table 6-2 above show some of the questions and the average answer numbers on the scale. Table 6-1 shows questions where the higher scale answer is better (more user satisfaction) while Table 6-2 shows questions where the lower scale answer is better in both diagram types.

6.1.4 Discussion

A higher percentage of constraints have been defined correctly using CSBE than using a wizard. A statistically significant difference appeared between the two techniques in both diagram types. The explanation of this result may be ascribed to

the ability of the users to express constraints visually more easily (using the visual diagram components) than expressing them through filling a form of properties, especially when URI references are used. Anecdotal evidence supporting this idea is the observation that the users, while using the wizard, drew example constraints, especially connection constraints, using pen and paper before starting the definition process. This suggests that visualising the constraint provides better (closer to the mind) understanding for its concept, which supports using CSBE and explains its performance in the constraint definition task. This also agrees with the claims of Bimbo & Vicario (1995) and Draheim et al. (2010) that visualisation increases the intuitiveness of specification. The feature of expressing constraints using the same visual elements (vertices and edges) of the target language has the main effect of increasing the intuitiveness according to Draheim et al. (2001). Many users made comments that support the idea of intuitiveness and the idea of reducing the gap between the formats of specification and application of constraints. A number of comments referred to the intuitiveness to the constraint visualisation and using the same visual elements of the target language in building the constraint examples, such as: “I like the process of inference gives all the possibilities and I like visualising the constraints”. “I did not like the wizard because it is not visual”. “I like the visualisation, it shows you the real constraint”. “Really more intuitive and simplifies complex cases definition compared with wizard”. A repeated comment was “You can see the constraint before saving it, so you are sure of what you have defined”. This is related to the synergistic approach which gives the user a list of different alternatives and the user selects from the list. Apparently, the synergistic approach implemented in DECS contributes to its intuitiveness. This is because the user interaction with the system by introducing an example to express the constraint visually and the reply of the system with a set of inferences allows the user to see all the different interpretations of the example. When the user interacts by selecting the required constraint from the list and confirms the selection, this appears to give the user confidence that the specified constraint is the required one. This confidence was limited in the case of using the form-filling technique since the user was not sure if the required constraint has been specified correctly or not.

In general, the participants required less time to define the constraints using CSBE than when using the wizard. This result can be explained by mentioning that

the wizard, as any form-filling technique, needs more time in reading the property labels and exploring the alternative values. This time is reduced in the case of CSBE. On the other hand, the CSBE technique consumes the user's time in thinking of suitable example(s) to express the constraint. However, CSBE outperformed the wizard in time measurement. It can be argued that with experience, properties in the wizard can be memorised while the time for expressing a constraint using CSBE will stay constant which may overturn the experimental result. The answer here is that observations and qualitative interviews showed that the user gets experience in CSBE quicker than the wizard (Table 6-1). Although it is not possible to generalise these results over long system usage and a large number of constraint definition tasks, they suggest that users can get experience in CSBE even if it depends on introducing different examples for each constraint.

As shown in the survival analysis, there was no significant difference between the two techniques regarding the time required to accomplish the task in the case of STD. By contrast, a strongly significant difference appeared in the case of UCD for the same measurement. It is difficult to explain this data. As mentioned above, the significant difference can be justified as the CSBE technique is easier than wizard and it is possible to accomplish the task in less time. This is clear from the curves in both diagram types. Constraint lists provided as tasks cannot clearly justify the difference between the diagram types because the constraints were of the same types. However, the individual constraints were, of course, different and may have different levels of difficulty. This difference in results between the two diagram types requires more investigation.

Regarding user satisfaction, users prefer the CSBE technique and were more satisfied using it compared to the wizard. Table 6-1 and Table 6-2 are self explanatory; users felt less stressed, less rushed, less uncertain, and more confident while performing tasks using CSBE. This might explain the better (higher) performance in correctness and time especially as they believed that they have done better in CSBE in a question not included in the tables. Users agreed that both techniques were powerful enough to define constraints with insignificant advantage to CSBE ($p = 0.33$ and $p = 0.19$ for STD and UCD respectively). Some questions in the after-experiment questionnaire asked which technique was easier to be learned and

remembered, which technique will you use for constraint definition if you were assigned such task, and which technique will be more effective? All their answers were to the advantage of CSBE.

The results presented above demonstrate that using CSBE for constraint definition in the context of meta-CASE tool reduces the complexity of the constraint definition task. Referring to the research problem introduced in Chapter 1, CSBE reduced the complexity because:

- It reduced errors associated with the constraint specification task.
- It reduced the time required to accomplish the constraint specification task.
- It reduces the gap between the constraint application domain and constraint specification formats. This is done through visualisation.

This indicates that the CSBE technique can add value to the meta-modelling process and to the CASE specification process in general. Consequently, the study rejects the null hypothesis and accepts the alternative one.

Because the same constraint suggestions can appear for different examples, participants came to expect the suggestion would appear in the same place. For example, constraints that deal with labels always appear at the bottom of the list. So users were starting to search the list from the bottom instead of starting from the top based on their experience during the training time. This suggests that it may be problematic to allow an inference engine to rank (and hence list) the suggestions based only on their likelihood in a particular context.

6.1.5 Threats to Validity

Limiting the task time to 15 minutes could have an effect on the correctness measurement. If subjects were permitted to continue without time limitation, some of them possibly would have achieved higher correctness. However, extending the time limit would not threaten the validity of the significant result for UCD as all subjects completed the task within 15 minutes using CSBE. It seems highly unlikely that extending the time limit for STD would produce a significant result but this needs further investigation. Intervention (mentioned in Section 6.1.2.5) also threatens the validity of the results. However, it only occurred in the form-filling condition which

worked against the CSBE technique. Since the CSBE technique outperformed form-filling, this indicates that the intervention had very little or no effect. Subjects' native language is also a threat to validity as not all the subjects are native English speakers. However, all the participants had or were carrying out their studies in English. Threats to validity of this research also include using only two diagram types (UCD and STD), which may limit the generality of the results. The decision to use two diagram types was taken to reduce the threat of bias compared to using only one type. Additionally, time and resource limitations prevented investigation of more diagram types.

6.1.6 Related Work

Although some empirical studies have been conducted in the context of meta-CASE tools and constraints, such an experiment to evaluate the CSBE technique (or PBE technique) against another technique has not been documented in any research before. One experiment (Offen, 2000) was conducted to evaluate the effect of constraints on the work of designers in diagram editors. To conduct this research, a meta-CASE tool, "CASEMaker", was built to generate constraint-dependent diagram editors. Results showed that extensive constraint messages from the system during the work of the designer reduce his/her novelty. However, such results cannot be compared with the one generated from this research.

An interesting study was conducted by Myers (1993) for evaluating the PBE system Peridot. The aim of the study was to evaluate how efficient Peridot is to use. Ten people, half of them programmers and half not, participated in the experiment. Time for building a menu task was calculated and results showed that Peridot is efficient as it reduces the time required to build a menu from 50-500 minutes to 4-15 minutes. The time spent to accomplish the task was also considered as a criterion in an empirical study conducted by Maulsby & Witten (1993) to evaluate the learning ability of the PBE system Metamouse. Quantitative and qualitative results show that Metamouse needs enhancement because it could not infer and learn the required repeated task from the users.

6.1.7 Conclusion

This section has presented an empirical study comparing two constraint definition techniques, a typical form-filling technique represented by a wizard and the Constraint Specification by Example (CSBE) technique. The latter has not been used in the context of meta-CASE tools before. Both techniques have been implemented in DECS and the study evaluated both techniques using two diagram types, a State Transition Diagram and a Use Case Diagram, with respect to constraint definition correctness, required time to accomplish a constraint definition task, and user satisfaction. Results show that, in general, the CSBE technique outperforms the wizard with respect to all the measured criteria. A general conclusion is that CSBE succeeded in reducing the complexity of constraint specification task by reducing the error associated with this task, reducing the time required to specify constraints and reducing the gap between the specification and the application formats through visualising the task and increasing its intuitiveness.

6.2 Constraint Polarities in Natural Language (Study Two)

6.2.1 Introduction

Expressing a constraint in a natural language and expressing the constraint in the system using a diagram example are two different things. In the natural language, different designers may describe a constraint differently according to their way of expressing it when talking to each other. However, in DECS, the expression of the constraint is done using a diagram example that has no relation to how the designer expresses it linguistically as shown in Chapter 4. As an example, the constraint expressed in Figure 4-11, although it is expressed negatively in English by starting with words “*it is not allowed...*”, it is easier to express in DECS as a positive example (Figure 4-11). In English, this constraint can be expressed positively as “*The lower bound number of Actor is 1.*” or “*At least one Actor must exist in the diagram*”. To explore this area of natural language constraint expression, an online survey was conducted before conducting any other study related to the polarity of examples in DECS. The following section describes this study.

6.2.2 Constraint Polarity Survey

6.2.2.1 Aim

An online study was conducted with the aim of studying the preference, in terms of understandability, of users for positive and negative descriptions of constraints described in English. The study consisted of 10 constraints that apply to State Transition Diagrams (STD). STD was used because it is well known to people in the Computing Science community.

6.2.2.2 Experimental Design

As Table 6-3 shows, a within-subject design was adopted for this experiment to deal with individual differences between participants. Each constraint was written in English in positive and negative forms. Subjects were asked to choose the easier to understand between the positive expression and the negative expression or both if they are equally understandable. Constraint expression polarity was considered as the independent variable with three levels, positive, negative and equal while the

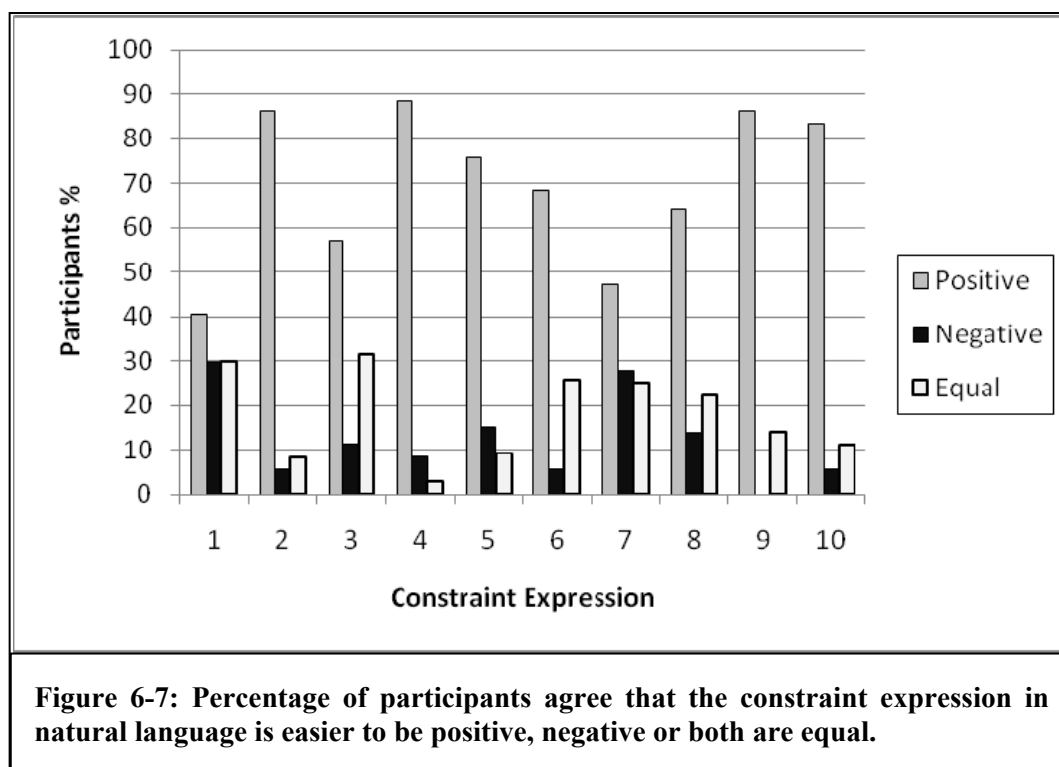
dependent variable was the number of participants that preferred (found it easier) one expression to the other. The evaluation was designed in the form of a questionnaire on a website and participants were invited by email. The questionnaire is available in Appendix D. Participants included people of different cultures, different relation such as friends (researchers at Bradford University), and different levels of Computing Science background including undergraduate students and researchers. Data was collected automatically by the survey tool SurveyMonkey (SurveyMonkey, 2011) used to perform the study.

Table 6-3: Natural language constraint expression preference study summary

Property	Value
No. of participants	37 (not all of them answered about all the constraints).
Adopted design	Within-subject
The independent Variable (IV)	Natural language constraint expression
No. of independent variable levels	3 (positive, negative, both)
Dependent Variable (DV)	Preference of one level of IV.

6.2.2.3 Results

Results showed that in all the constraints, expression using positive language was preferred over negative as shown in Figure 6-7. Since some of the participants did not evaluate all the constraints provided in the questionnaire, the percentage of the participants that preferred one polarity over another was considered instead of the actual number. Because the number of participants was over 30 in all the expressions, Central Limit Theorem (CLT) is applicable which assumes normality of data. Accordingly, a t-test was used to analyse the differences between constraint expression results. A t-test was used for individual comparison as a safer solution than using ANOVA which is used only for between-subject design.



Results show that there are significant differences between positive and negative expressions and also between positive and equal with p-value ($p < 0.001$) for both comparisons. On the other hand, there was no significant difference between the negative and equal selections ($p = 0.104$).

6.2.3 Discussion

These results demonstrate that people may understand and prefer one expression polarity of the constraint over another. In particular, subjects significantly preferred and understood constraints that are written in positive natural language expressions compared to those written negatively.

Based on these results, with the fact that there is a significant difference between the preference of positive and negative constraint expressions in natural language, it was decided that the natural language expression of a constraint should be taken into consideration as a potential confounding factor in any study or experiment that focuses on studying constraint polarity. Accordingly, this confounding factor must be counterbalanced to avoid its threats to study validity.

6.2.4 Threats to Validity

The threats to validity of this study can be summarised as follow:

- As an online study, the study was conducted away from the eyes of the researcher. This cannot guarantee the control of the study such as the time spent in conducting the experiment or the care given in answering the questions. As a trial to avoid this threat, the researcher invited trusted friends to avoid answering the questions randomly without giving care to the study in addition to the undergraduate and postgraduate students and researchers at Glasgow University.
- Some of the participants did not finish the study (did not answer all the questions) which may threaten the results and its analysis. Another related issue is that the study could not determine the reason behind not completing the online questionnaire. This issue was handled by analysing the percentages instead of the absolute numbers as shown above in the results presentation.
- As has been introduced in Chapter 5, using friends in this study with the aim to increase the sample size, may threaten the validity of the results. However, they were not told about any expectation or preference towards specific results.

6.2.5 Conclusion

This section introduced an online study as part of this research with the aim to discover the preference in terms of understandability towards a positive or negative polarity of natural language in expressing constraints. Results showed that participants preferred positive language over the negative one. This study has also the aim of helping to design the next study (Study Three). Results suggest that the polarity of natural language used in expressing the constraints is a confounding factor that needs to be taken into account in the design of Study Three.

6.3 Example Polarities (STUDY THREE)

6.3.1 Introduction

The results of Study One showed the superiority of CSBE over a typical form-filling technique in constraint definition (Qattous, Gray & Welland, 2010). However, the DECS version that was used for conducting this experiment did not allow the user to specify the polarity of his/her example explicitly. This means that the user provides the example and asks the system to infer from the example. The system infers both polarities and presents them in one list. So the users were not given the choice to specify explicitly the required interpretation of the example, either positive or negative. Users were trained and told about the example polarity feature and asked to express constraints using a suitable example with a suitable polarity. From their comments, documented in the post-experiment questionnaire, it was clear that the users understood the example polarity principle quickly and they applied it easily (i.e., they did not demonstrate any noticeable effort in considering how to be able to apply it).

However, although DECS users liked the ability to express constraints using both polarities, positive and negative, there was no evidence that allowing such a feature would influence the performance of CSBE. Some users commented that they would have liked to have been able to select an example polarity explicitly, rather than keeping it as an implicit feature. Inferred constraints was another issue. The constraint list generated by DECS contained together both positive and negative interpretations of the example. Some participants in Study One commented that they were confused by this. In addition, this also increases the number of inferred constraints presented in the list, which may increase the time required to search the list for the required constraint.

As a result of these issues, the research raised a second question, *“Does example polarity influence the performance of CSBE?”* In other words, *“Does allowing constraints to be expressed using multi-polarity examples affect the performance of the CSBE technique?”* In particular *“Does using multi-polarity examples add value to the performance of CSBE compared to using uni-polarity*

examples?” To answer these questions it was decided to conduct a study to evaluate the performance of CSBE when allowing constraints to be expressed using different polarity examples compared to restricting example expression to only one polarity. In general, the study aimed to determine the more suitable implementation that improves the performance of CSBE technique by evaluating two different DECS implementations. The first allows the constraints to be expressed using different polarity examples and the second restricts the user to only one example polarity. Since the study focussed on example polarity, it was decided to allow the user to specify example polarity explicitly. Explicit example polarity specification means that there will be a control by the user over the inference engine to direct it either to infer and interpret the example positively or negatively. This agrees with the recommendations of the users in Study One and handles the long list issue introduced in the previous paragraph. Explicit example polarity specification was discussed in Chapter 4. However, it had not been implemented yet in the DECS version used for the experiment discussed in 6.1 and, therefore, was not used in Study One.

Since this experiment was related directly to the example polarity, there was a major potential confounding factor that could threaten the study, viz., the polarity of the natural language used to describe the constraints in the experiment. Accordingly, results of Study Two were taken into consideration as they contributed to the design of Study Three and helped in reducing the threats that can result from the natural language constraint description in the tasks given to users.

Although the example polarity feature in DECS has been discussed before, this section starts with a brief recall for what has been discussed in Chapter 4 with some helping examples. The section then describes the different components of the two conducted experiments with explanation of their design and execution. Experimental results are presented, analysed, discussed and compared to some different related works. The section ends by concluding from the studies conducted.

6.3.2 Positive and Negative Examples in DECS

To recall from Chapter 4 a positive example represents the desired state or what must hold in the tool to be generated, while a negative example represents an undesired state or what is not allowed. In Chapter 4, some examples were introduced

to explain this constraint example feature implemented in DECS. As a reminder example, the constraint “*it is not allowed to have less than one Actor in the diagram*” (or “*the lower bound number of Actor is 1*”). This example is shown in (Figure 6-8). The same example with the opposite interpretation, negative, is shown in Figure 6-9.

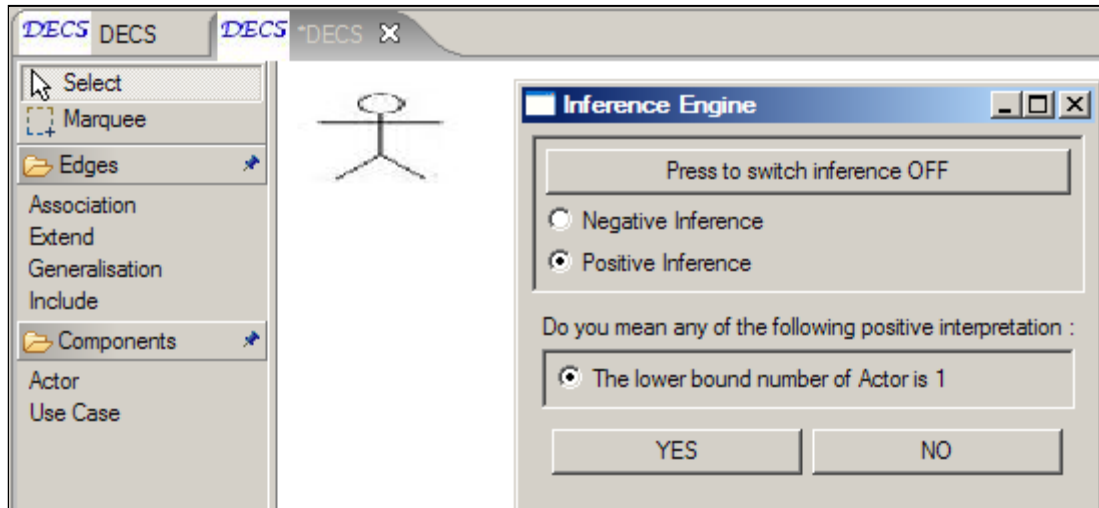


Figure 6-8: Positive interpretation (inference) the introduced example.

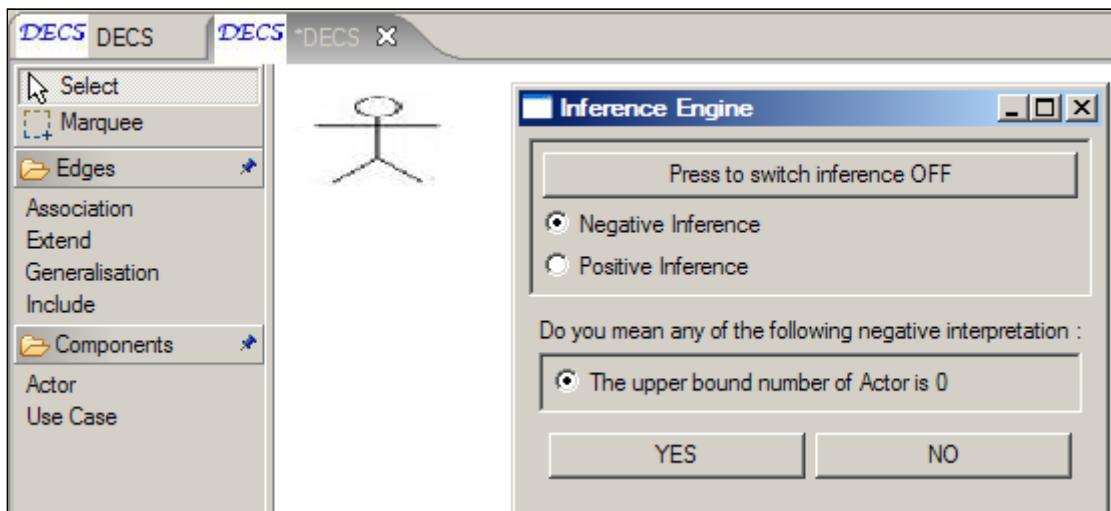
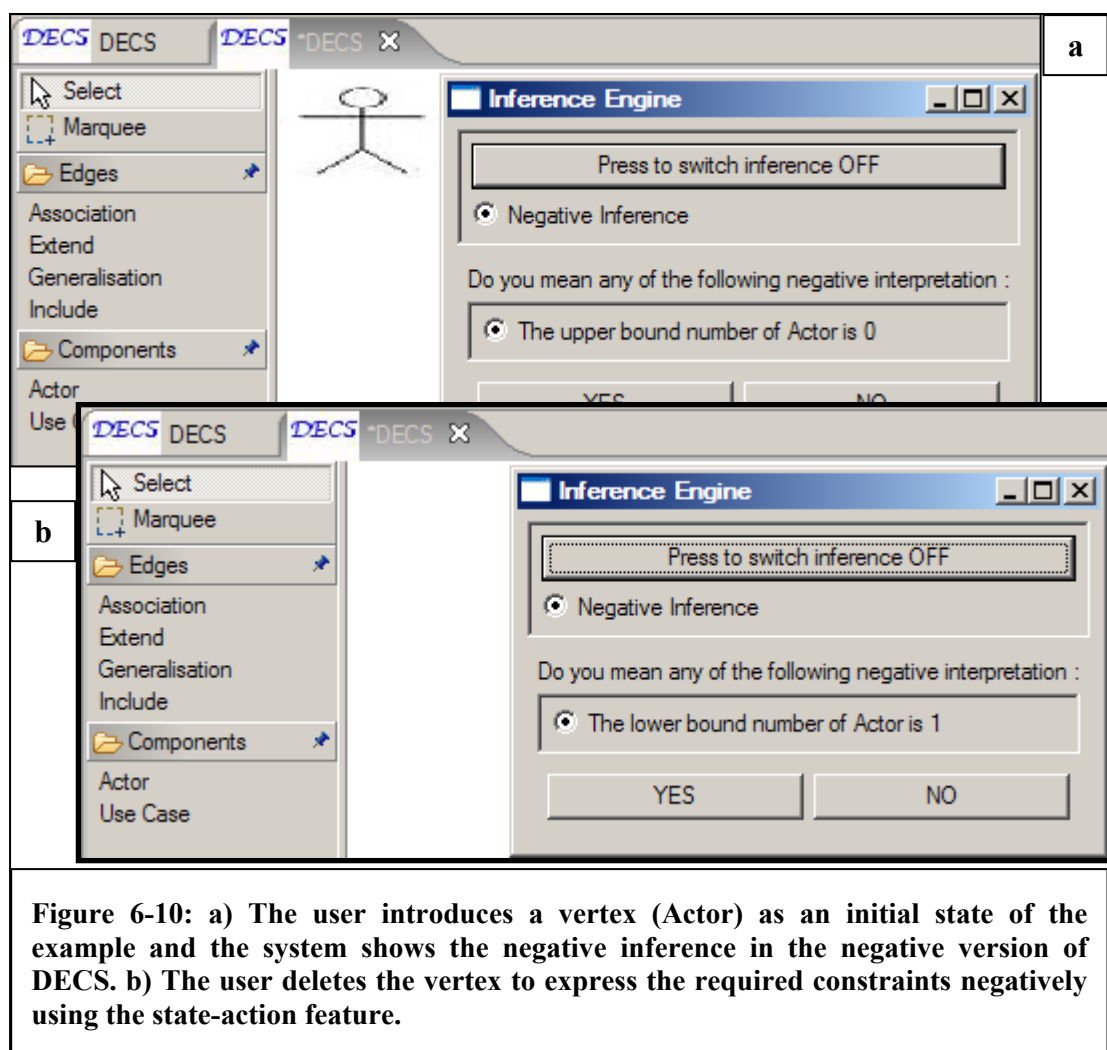


Figure 6-9: Negative inference for the same example introduced in (Figure 6-8).

6.3.3 Inference Over State and Action

Recall from Chapter 4, DECS can infer from the example state and an action. The following example is repeated from Chapter 4 for reminding with the state-action feature. The constraint (expressed in Figure 6-8) can be expressed negatively instead

of positively using the state-action technique. Figure 6-10-a shows part of the example which is composed of one Actor. Negatively, this is interpreted by DECS as the constraint “*It is not allowed to have any Actor in the diagram.*” (or as the figure shows “*The upper bound number of Actor is 0.*”). This part is considered as the state part of the example. The action part is when the user deletes the Actor vertex (Figure 6-10-b). Taking the previous state of the example into consideration, the system can infer that the required constraint is “*It is not allowed to have less than one Actor in the diagram.*” (or as the figure shows “*The lower bound number of Actor is 1.*”) (or, “*There must be at least one Actor in the diagram.*”) as shown in Figure 6-10-b.



6.3.4 Effect of Example Polarity

6.3.4.1 Aim

This study aims to discover the effect of example polarity on the performance of the CSBE technique. In particular, the experiment studies the effect of allowing the user to express the constraints using multi-polarity examples compared to uni-polarity examples. This is conducted by comparing two different DECS versions for constraint expression using example(s).

6.3.4.2 Experimental Units, Materials, and Tasks

To achieve the aim of the study, there was a need to compare tools of different polarities. The first step was to determine the different possible implementation alternatives for different polarities. The following table (Table 6-4) summarises eight recognised implementation alternatives for constraint example polarity:

Table 6-4: Possible implementation alternatives for example polarities.

Possible alternatives	Symbol
Negative only supported with state-action feature.	$N + A$
Negative only without state-action feature.	$N - A$
Positive only supported with state-action feature.	$P + A$
Positive only without state-action feature.	$P - A$
Negative with state-action and Positive with state-action.	$(N + A) + (P + A)$
Negative with state-action and Positive without state-action.	$(N + A) + (P - A)$
Negative without state-action and Positive with state-action.	$(N - A) + (P + A)$
Negative without state-action and Positive without state-action.	$(N - A) + (P - A)$

Among all the alternatives above, it was practically possible to use only two choices of different tool (DECS version) implementations for the study. The limit on the number of alternatives arose for the following reasons:

- **Time problem:** the time required to conduct the experiment with tasks for all the possible tools would be long, not counting the time required for training the participants on using them. One practical problem this research faced was finding participants who would agree to do an experiment that is 60-90 minutes long. This is actually the time required for testing two conditions. Based on this, conducting all the alternatives or more than 2 of them would not be feasible and would have exhausted the participants, assuming they could be found.
- **Aim problem:** the aim of the study would be altered as it specifies comparing between the uni-polarity compared to multi-polarity tools. Consequently, comparing other tool alternatives is not in the scope of this study.
- **Design and data analysis problem:** Comparing the different conditions or even more than two of them would complicate the experiment, its design and its data analysis. One of the problems faced in this experiment particularly is its complex design for counterbalancing the different variables. Additionally, the statistical analysis of the data would be complex as the comparison would be between different conditions with each other instead of concentrating on two only.

Between all the choices above, the following two different tools implementations were selected as conditions to conduct the experiment on:

- **The Negative without state-action and Positive without state-action (N-A) + (P – A) or the Negative Positive tool (NP tool hereafter),** which offers interpreting the examples either positively or negatively according to user choice. In other words, this tool can infer from the negative and positive examples but without support for state-action feature.
- **The Negative only supported with state-action feature (N + A) or the Negative Action tool (NA tool here after),** which interprets the examples negatively only. However, the state-action feature was also implemented in this tool to support negative inference so the tool can infer all the required constraints. In other words, this tool can infer constraints from the “negative and action” examples.

These two implementations were selected for the following reasons:

- They are sufficient to test the hypothesis and achieve the aim of this study as they include the required multi-polarity and uni-polarity alternatives.
- It was fair to compare the two selected choices since neither the negative nor the positive tools without action support ((N - A) and (P - A)) would be expressive enough to express the constraints. Because of that, these two options were eliminated. Using the negative and positive supported with action tool ((N + A) + (P + A)) was also not fair to be part of the comparison because this accumulates *all* the features in one tool leaving nothing to compare with. Accordingly, this alternative was also excluded. There are two additional unbalanced options that were eliminated, viz., (N + A) + (P - A) or (N - A) + (P + A).
- Selecting these two tools offered the options that allowed a comparison of polarity exploring the added value of including positive examples since the negative option is included in both tools.
- The positive with action tool (P + A) was not selected because of the conceptually negative nature of the constraints and because the action concept was clearer in the negative tool than the positive one for expressing the constraints. This was observed during preliminary and pilot studies. The pilot studies aimed (i) to help decide the suitable polarity to be used as a representative in the uni-polarity condition and (ii) to estimate the required time and the difficulty of the experiment. During these studies, and from the discussion conducted before them throughout the training time, it was noticed that it was easier to explain to users the action concept with negative examples than with the positive examples. One more reason for not selecting the (P + A) tool was the problem associated with the positive (state-action) examples mentioned above (Figure 4-12-b). This made using the negative tool a better (higher performance) choice to avoid any confusion for users while trying different examples using the state-action feature.

It is believed that the selected tools offered, on balance, the most expressive and realistic alternative for each of the two conditions and, consequently, it was decided to choose the NA tool as a representative for the uni-polarity condition and NP as a representative for the multi-polarity condition. Additionally, both selected implementations are powerful enough to express a wide range of constraints. They also depend on two different backbone concepts of expressing the constraints using examples, as one depends on different polarities while the other depends on one

polarity supported with the action concept. Both tools have almost the same GUI as shown in Figure 6-8 and Figure 6-9 for NP tool and Figure 6-10 and Figure 4-10 for NA tool.

The study required participants to carry out a set of supplied constraint specification tasks that partially define a modelling diagram type. Two diagram types were selected: State Transition Diagram (hereafter STD) to conduct the experiment on and Use Case Diagram (hereafter UCD) for training and experiment explanation purposes only. These diagram types were selected for the same reasons indicated in the first study in the previous section.

The experiment was conducted in Scotland at Glasgow University. Sixteen participants were selected from Computing Science, Information Technology, and Software Engineering postgraduate students. An invitation was posted to all the postgraduate students by email. Students who replied with a completed consent form were selected. Each participant was paid £10 for participation. The selected subjects had no experience or previous knowledge of the tools to be tested; however, they were required to have knowledge of STD and UCD. No knowledge of the constraints or their definition was required. These conditions were mentioned in the invitation email that also contained information about the general aim of the study.

To prepare the tasks for the users, a main constraint list was created for STD. The list contained constraints that define the diagram language. Some other constraints were added to the list as customised constraints that define potential customised requirements to take advantage of the power of DECS as a meta-CASE tool. Constraints in the list were divided into six different groups based on the following criteria:

- Constraints related to cardinality of vertices in the diagram. This class contains constraints that define the upper bound and lower bound numbers of allowed vertices such as the constraint *“It is not allowed to have more than one Start State in the diagram”*.
- Cardinality of incoming and outgoing edges such as the constraint *“End State can only have incoming edges”* or *“The cardinality of outgoing edges from End State is 0”*.

- Unique visual representation of vertices such as “*In any given diagram, Start State must have a unique visual representation*”.
- Vertices label-related constraints such as “*Non-transition States must have labels*”.
- Edges label-related constraints such as “*Outgoing Transition edges from Non-Terminal States, must have unique labels*” and “*Transition edges must have labels starts with the substring ‘out’*”.
- and, the path related constraint, “*There must be a path between Start State and every other vertex in the diagram*”.

From each group two constraints were selected and each selected constraint was placed into a separate list. Two groups, “*unique visual representation*” class and the “*path*” related class, had only one constraint in each group, apart from the vertex and edge types; consequently, very similar constraints of the both groups were used in both lists. At the end of the process, two lists each with 6 constraints, each constraint from a different group, were generated. Using the above method of constraint selection, both generated constraint lists contained 6 similar, but not identical, constraints and each list formed a *task*. The tasks (constraint lists) showing the different constraints used appear in Appendix E.

Each constraint of the 6 in each list has a predefined (implemented) logical example that expresses it. In the NP tool, each list had 4 constraints out of the 6 that can only be expressed (specified) using positive examples (constraints number 1, 3, 4, and 6 in Appendix E). Similarly, in the NA tool, 4 of the 6 constraints can only be expressed using negative with state-action examples. Each list of the two was assigned to a tool and the assignment was counterbalanced. Based on Study Two results, and to avoid the natural language effect discussed above, one of the lists was written starting with a negative language for the first constraint, positive language for the second one and so on. By contrast, the second list started with a positive language for the first constraint. Since the two lists contain similar but not identical constraints, the lists also were used for half of the participants. For the other half, the lists were swapped, meaning that the list that started with a negative constraint for the first half of the participants, was replaced by one with positive for the second half of the participants.

These precautions guaranteed a counterbalanced assignment of the constraints to the participants and avoidance of the natural language effect. Participants were assigned to the task according to Table 6-5 which summarises the distribution and counterbalancing precautions. The table shows the condition that each user starts with since each user performed both conditions. For example, user 1 starts with the NA tool using positive language list and the second part of the experiment was the NP tool with the negative language list.

Table 6-5: Design of the experiment and user first assignments to different counterbalanced conditions.

16 Participants			
Negative-Action Tool (NA)		Positive-Negative Tool (NP)	
Positive Language	Negative Language	Positive Language	Negative Language
User 1	User 2	User 3	User 4
User 5	User 6	User 7	User 8
User 9	User 10	User 11	User 12
User 13	User 14	User 15	User 16

Each participant was asked to perform two tasks, each using one of the two different tools (NA and NP). Each task was to define the 6 constraints in the list provided using one of the tools. As a part of the task, the user was also asked to answer several questionnaires at different stages of the experiment. All the questionnaires used are available in Appendix E.

Three pilot studies were conducted for the purposes described above. During the pilot studies the time required to finish the tasks was estimated so participants can be interrupted if not accomplishing the tasks within the time limit. Later on it was decided not to interrupt the participants; instead, there was no time limit because that was easier and more convenient for the resulting statistical analysis.

6.3.4.3 Variables and Hypothesis

The independent variable of the experiment was tool implementation, which was of two levels (the two implementations) represented by the two tools, Negative tool with state-action feature and Positive-Negative tool. The following dependent variable measurements were tested for each implementation (tool).

- The *effectiveness* in terms of the resulting constraint specification **correctness**.
- The *efficiency* in terms of the **time required** for accomplishing the constraint specification task.
- The *user satisfaction* with the technique.

6.3.4.4 The Hypothesis:

The null hypothesis of this experiment states that,

H0: *there is no difference between the two implementations regarding the variables to be measured.*

The alternative hypothesis states that,

H1: *there is a difference between the two implementations regarding the variables to be measured.*

6.3.4.5 Experimental Design and Execution

For this evaluation, a **within subject** design was adopted as each participant was asked to perform two tasks, each task using one of the tools. This design was selected because of its simplicity in application and analysis. Adopting this design allowed also the avoidance of the affect of personal variations between different subjects, such as experience, IQ, etc.

Each participant was trained before each experiment, using UCD. Participants' training was conducted using different constraint examples and a different diagram, UCD, to avoid any threatening effect of the training on the experiment itself. The training included introducing subjects to the concepts of negative example, positive example, and the action notion using a white board and tool demonstrations. The participants were encouraged to use the tool freely and they

were supported with all the required assistance to be familiar with it. Training sessions took about 10-15 minutes each.

Each task was conducted in 6 steps representing the 6 constraints of each task. The participants were given the constraints to define one at a time, which helped in focusing on the current constraint without looking to the previous or next constraint in the list. Each constraint definition was recorded separately by screen capture to be used later for data extraction and analysis. Deleting the previous example and starting with the new one was taken as an indication of starting the constraint definition while pressing the save button, to save the constraint definition, was taken as an indication of finishing the constraint definition. This helped in measuring the required time for each constraint. Participants were instructed at training time to choose any answer and finish the constraint definition to indicate giving up if they could not reach the required constraint.

After each constraint definition the participant was asked to fill in a *post constraint* questionnaire. Once the task of 6 constraints was completed, participants were asked to fill in a *post task* questionnaire, in order to evaluate each tool. After finishing both tasks on both tools, participants were also asked to fill in a *post experiment* questionnaire, followed by a short interview to compare the tools.

It has to be mentioned here that one of the constraints, the visual representation related constraint, was eliminated from the results because of an implementation bug. This bug can be summarised as follows: this constraint designed to be expressible using a positive example in the NP tool and was also designed to be expressible using a state-action example in the NA tool. This constraint was one of the 4 constraints out of the 6 used in the experiment (mentioned before in Section 4.2) which was implemented to be inferred from a positive example in the NP tool and using a state-action example in the NA tool (constraint number 3 in Appendix E). This was provided the required controlled condition and counterbalance of always having 4 constraints in each list either expressible using positive examples in the NP tool or using negative examples with the state-action feature in the NA tool. It was also part of the design of the experiment. However, because of an implementation bug, it was possible to express the constraint using a negative example with or without the state-action feature in the NA tool. By contrast,

it was only possible to express it using a positive example in the NP tool. This created an unequal number of constraints that use negative examples and positive examples in the NP tool and constraints that use negative examples and negative examples with state-action in the NA tool in the two lists for each user. After the results analysis, it has been discovered that this bug may have compromised the design; however, it did not compromise the revised design when the constraint is removed from the analysis. Accordingly, all of the results of this constraint were removed from the analysis and the results presented here depend only on the revised design with the problematic constraint eliminated. Thus, only 5 constraints' data were analysed and are presented. After the elimination of the visual representation related constraint, only 3 constraints were left that could be defined either positively in the NP tool or negatively with the action-state feature in the NA tool. The other constraints in each list, 2 constraints, were defined negatively in the both tools. Since this experiment design is within-subject, no normality assumptions were made about the collected data. Based on this, the nonparametric Wilcoxon Signed-Rank test was used to analyse correctness, time required to accomplish the tasks, and satisfaction results.

6.3.4.6 Results

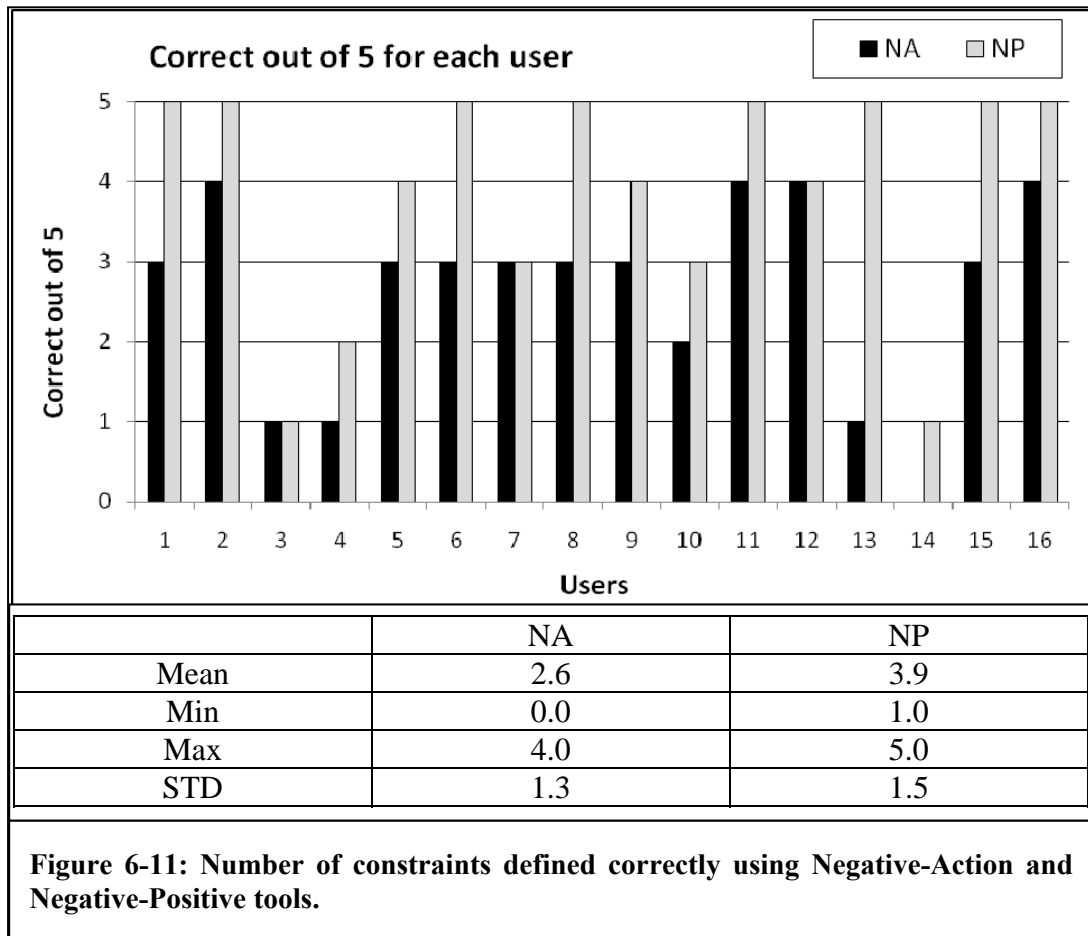
Most of the participants indicated that they were familiar to some degree with the constraint definition task with an average of 3.0 on a scale of 5.0. Each experiment (participant) took 60 - 90 minutes including training and questionnaire filling time. The comparison between the two tools regarding the required measurements is presented in the following sections.

6.3.4.6.1 Correctness

The number of correctly defined constraints for each participant was gathered from the recorded screen capture videos. All the constraints were attempted by all the participants. Figure 6-11 shows the number of constraints correctly defined by each participant and Figure 6-12 shows the number of participants who defined each constraint correctly. Both figures show the results for the NA and the NP tools.

6.3.4.6.1.1 Correctness for each participant

13 out of 16 (81.3%) of the participants defined a higher number of constraints correctly using the NP tool than the NA tool. None of the participants defined more constraints using the NA tool while 3 (18.8%) participants defined the same number of constraints correctly using both techniques. Half of the subjects, 8, defined all the constraints correctly in the task using NP, and none did using NA (Figure 6-11).



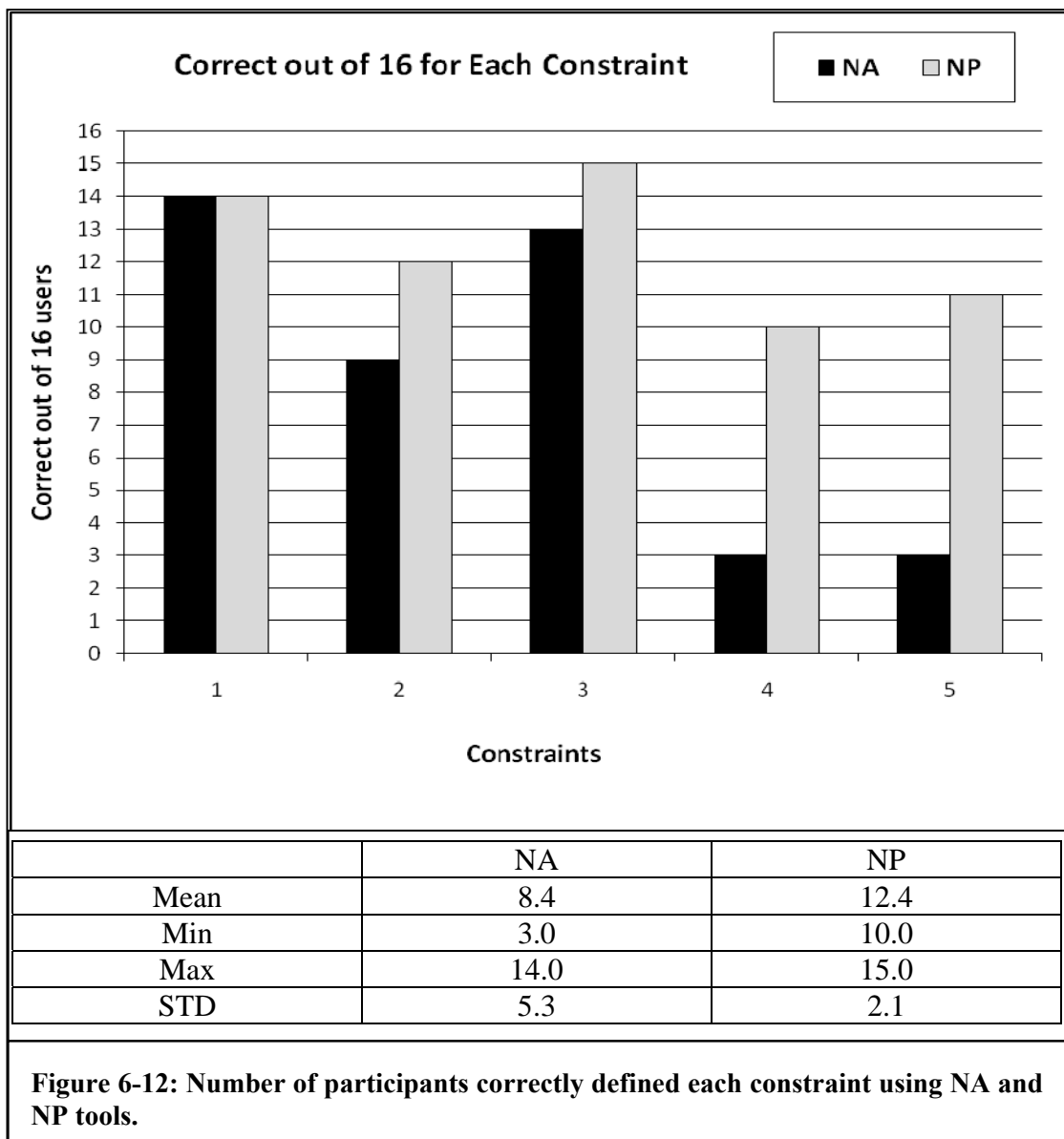
The average percentage of constraints defined correctly using the NA tool was 52.6% (2.63 constraints) while it was 77.6% (3.88 constraints) using the NP. Statistical analysis shows that there is a highly significant difference between the tools regarding correctness in constraint definition ($Z = -3.28$, $p = 0.001$).

6.3.4.6.1.2 Correctness for each constraint

4 constraints out of 5 (80%) were defined correctly by a higher number of participants using the NP tool than the NA tool. Only one constraint, the cardinality

related constraint, was defined correctly by the same number of subjects in both tools. The highest number of subjects, 15, was for the vertex label related constraint using the NP tool. The biggest difference between the two tools appeared in the last two constraints which are the edge label related constraint and the path existence constraint (Figure 6-12).

The average percentage of participants that defined constraints correctly using the NA tool was 52.5% (8.4 participants) while it was 77.5% (12.4 participants) using NP. Although the number of users that defined constraints correctly using NP tool is higher in 80% of the constraints than those using NA, statistical analysis shows that there is no significant difference between the two tools ($Z = -1.83$, $p = .068$).

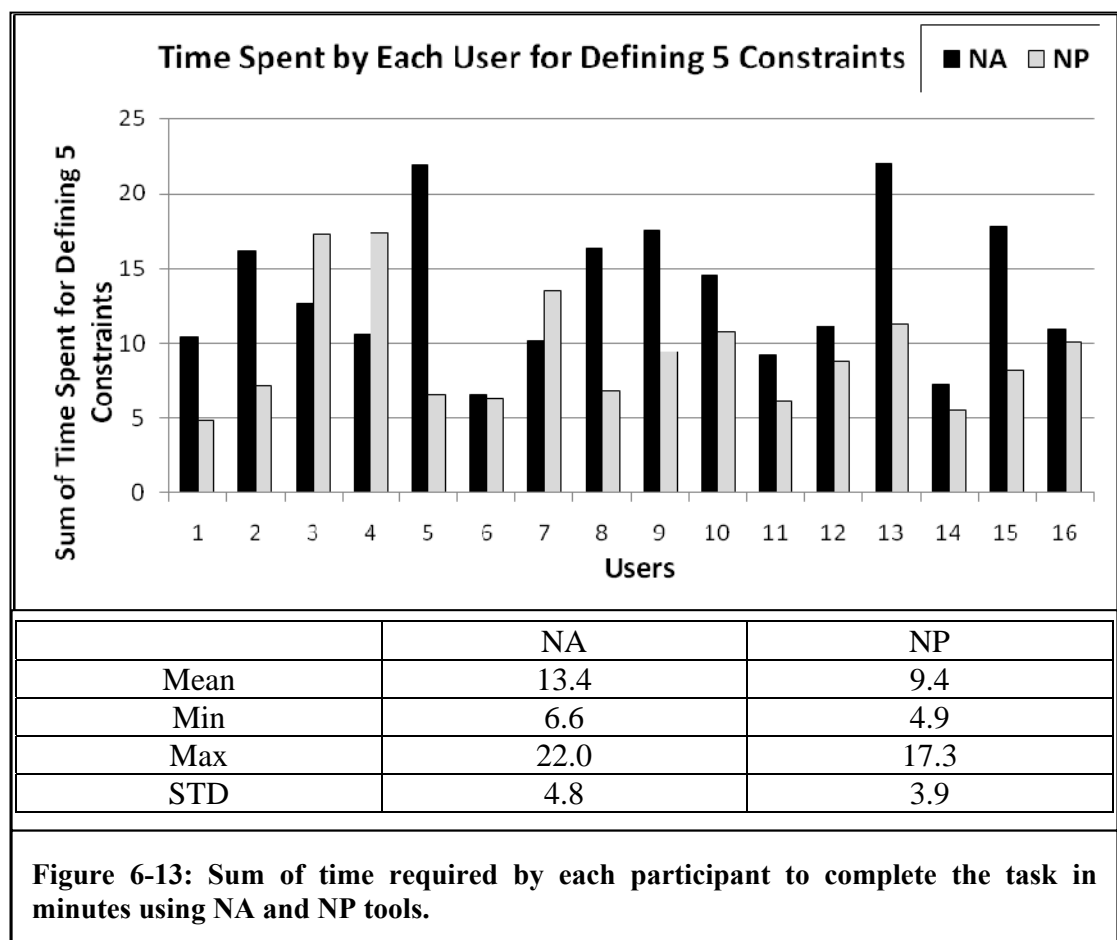


6.3.4.6.2 Time

The time required to accomplish each constraint definition was gathered from the recorded screen videos. To calculate the total time required for each task, the time periods spent by each user to define each constraint were summed together. Figure 6-13 shows the time required by each user to define the 5 constraints and Figure 6-14 shows the time required to define each constraint by all the users. Both figures show the time for the NA and the NP tools. In both diagram types NP tool required less time than NA.

6.3.4.6.2.1 Time spent by each user to accomplish the task

The time required to define all the constraints by each participant was summed to get these results. 81.3% (13 of 16) of users accomplished the task in less time using the NP tool while the rest of participants or 18.8% (3 of 16) of users accomplished the task in less time using the NA tool.

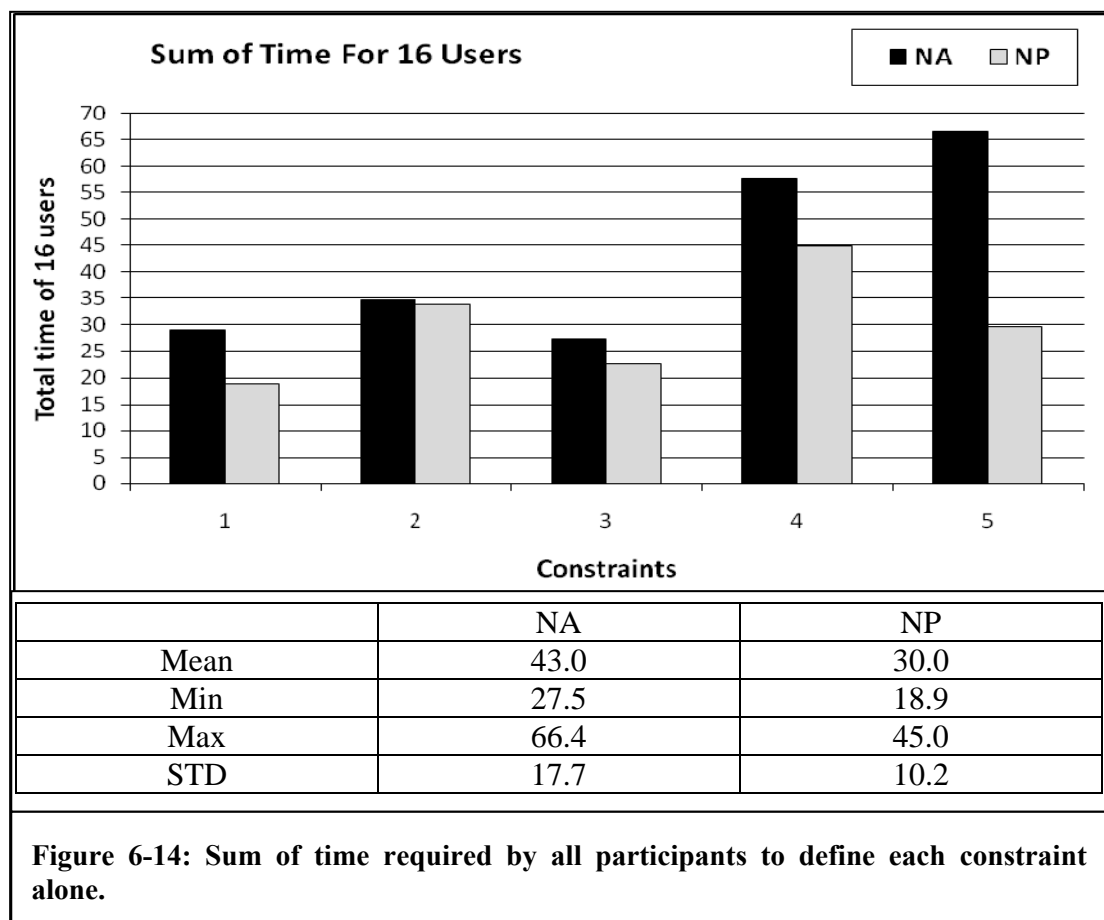


Two participants required more than 20 minutes to accomplish the task using NA, while all the participants required less than 20 minutes to finish the task using NP. More than double, 68.8%, of the participants finished the task in 10 minutes or less using NP compared to NA, 25%. The longest overall time record was 22 minutes using NA and the shortest time was 4.88 minutes using NP tool. Average time spent was 13.44 and 9.38 minutes for the NA and NP respectively (Figure 6-13).

Statistical analysis shows a significant difference ($Z = -2.28$, $p = 0.023$) between the two tools regarding the time spent by each participant for defining constraints. Apparently, the difference is to the advantage of NP tool as less time was required to accomplish the task using it.

6.3.4.6.2.2 Time spent to define each constraint by all the participants

The time required to define each constraint by all the participants was summed together to get these results. All the 5 constraints required a longer time to be defined using the NA than using the NP tool.



The longest time sum required to define a constraint was 66.4 minutes, for the path existence related constraint, using the NA tool. By contrast, the vertex cardinality related constraint required the shortest time, 18.9 minutes, using the NP tool (Figure 6-14). The average of time required by all users to define a constraint was 43 minutes and 30 minutes for NA and NP respectively.

As Figure 6-14 shows, the NP outperforms the NA tool because the constraints are defined using less time. Statistical analysis shows a significant difference between the tools ($Z = -2.02$, $p = 0.043$).

6.3.4.6.3 *User Perception*

In order to provide further validation for the hypothesis, post constraint, post task and post experiment questionnaires that participants filled out were analysed. In all the questionnaires there was a need to discover the feelings of the participants while interacting with each tool. A Likert 5-point scale was used and the order of some of these was inverted to reduce bias. Table 6-6 shows the average answers of users in the “post-constraint” questionnaire. Numbers in bold indicate significant difference between the tools.

Table 6-6: User perception of the NA and the NP tools in post-constraint questionnaire. (shaded = better (higher user satisfaction), bold and underlined = significant difference)

Questions	NA	NP
It was difficult to express the constraint with an example	<u>2.5</u>	<u>1.6</u>
It was easy to find the required constraint in the inferred constraint list	<u>3.7</u>	<u>4.4</u>
I was confident that I defined the required constraint	<u>3.7</u>	<u>4.5</u>
It was confusing to convert the English constraint expression to example	<u>2</u>	<u>1.7</u>
The way the constraint is written in English in the constraint list (the paper in your hand) affected my choice of the way I should express the constraint with an example	2.1	2.3

One aspect to be measured for user perception is the effect of the English language that the constraints were written in. As described before, some constraints were written in a negative form while others written in a positive one. Table 6-6 shows that there was a slight significant difference ($p = 0.041$) between the two tools regarding the confusion because of the English language. However, there was no significant difference regarding the effect of English on the way that the user expresses the constraint.

Table 6-7 and Table 6-8 show the average user perception answers of some significant questions in the “post-task” questionnaire. Table 6-7 shows questions where the higher scale answer is better (higher user satisfaction) while Table 6-8 shows questions where the lower scale answer is better (higher user satisfaction). Two tables are used because inverted Likert scale in the questionnaires was used.

Table 6-7: User perception of the NA and the NP tools in post-task questionnaire. (higher = better (higher user satisfaction), bold and underlined = significant difference)

Questions	NA	NP
How successful were you in accomplishing what you were asked to do?	<u>3.2</u>	<u>4</u>
In most cases, I achieved the required constraint at the first attempt.	3.9	4.3
Constraint definition task was easy using this tool.	<u>3.1</u>	<u>4.3</u>

Table 6-8: User perception of the NA and the NP tools in post-task questionnaire. (lower = better (higher user satisfaction), bold and underlined = significant difference)

Questions	NA	NP
How mentally demanding was the task using this technique?	3.4	3
How hard did you have to work to accomplish your level of performance?	3.2	2.5
Using this tool requires a lot of time and effort because I need to think of an example to express the constraint	<u>2.9</u>	<u>1.8</u>
How uncertain, discouraged, irritated, stressed, and annoyed were you?	<u>2.6</u>	<u>1.7</u>
While I was working, I felt that I needed help from an expert.	<u>2.9</u>	<u>2.1</u>
I was often unsure of what action to take next.	<u>2.6</u>	<u>1.9</u>

6.3.5 Discussion

A statistically significantly greater number of constraints have been defined correctly using the NP than using the NA tool. The statistical significance also appeared between the two tools, with the advantage of the NP tool, in terms of the time required to define the constraints. *From these results, it can be concluded that expressing constraints using negative and positive examples is more effective and efficient than when using negative examples only supported by the state-action feature.* This conclusion is supported by the results showing higher percentages of users defining constraints correctly within a shorter period of time using the NP tool than the NA tool. This gives a strong indication that spite of the apparently negative nature of constraints, it is easier, in some situations, to express them using positive examples. A close look at Figure 6-12 and Figure 6-14 confirms that two constraints

(4 and 5) were much easier to express positively as a lower number of participants defined them correctly and took more time than with other constraints using NA tool.

A note here is that although Study Two shows a significant difference between the positive and negative natural language for expressing the constraints, this does not show a significant difference in this experiment as appears in Table 6-6. This is because of the careful counterbalance of the language polarity of the constraints over each of the tools as previously discussed in designing the constraints lists.

User satisfaction results from the questionnaire after each constraint (Table 6-6) show clearly and significantly that the NP tool outperformed the NA tool on the constraint level except in one case and it does not show a significant difference. Expressing constraints and finding them in the inferred list using the NP tool was significantly easier since the positive interpretation list is separated from the negative one. A significant difference also appears in the confidence of the user that s/he defined the required constraint. It was significantly easier to think of an example for the constraint written in English using the NP tool. The English language polarity showed no significant effect on constraint expression in both tools. After-task questionnaires (Table 6-7 and Table 6-8) show that NP is significantly preferred by the users because they felt that they did better (higher performance) using NP, achieved the required constraint from the first attempt, and the constraint definition is easier using it. Users also felt that NA needs significantly more effort, it is less natural or logical as they felt the need of expert assistance when using it, and it is significantly more confusing since the users were unsure what the next step should be.

Observations, participants' answers in the exit questionnaire, and qualitative interviews showed that NP is preferred and considered more powerful for almost all of the participants which agrees with the task results. One participant only preferred the NA because he believes that the state-action feature is a powerful tool that increases the expressiveness and reduces confusion. By contrast, the rest of the participants supported the NP tool and agreed that the availability of the two example interpretations to express a constraint gives a bigger chance to express it in a more natural and logical way and provides more alternative solutions to look at. Although the users liked exploring different inferences generated by the opposite interpretations of the NP tool, it is believed that any implementation, including that used in the

current research, will aim to reduce the number of generated choices and inferences. This is to avoid overwhelming the user with inferences which may reduce the advantages of using CSBE. It also gives the user the feeling that s/he is dealing with an intelligent system that understands her/his examples; this was one of the comments of two participants documented in their interviews.

In NP DECS version, the GUI helps by separating the two interpretations into different lists. All the participants liked the transparency feature and the recorded screen capturing showed that all of them kept the inference engine “on” while working. These results also indicate the higher user satisfaction with the NP tool than NA. This concludes that users preferred using tools with different alternatives for expressing the constraints than sticking to only one polarity. In general, it can be concluded that example polarity affects the performance of CSBE in terms of affecting the effectiveness, efficiency and user satisfaction. The availability of positive and negative polarities to express constraints affects the performance of CSBE positively compared to the availability of only negative polarity supported with the action feature.

Participants were asked if they prefer customising the tool by learning the system so they can express some constraints using their own examples. All of them supported the idea, since many of them faced a problem in expressing constraints 4 and 5 (Figure 6-12). This supports a rules augmentation and customisation feature because they felt that they need a more “natural way” than already implemented to express the constraints.

6.3.6 Threats to Validity

One of the threats to validity of this study is time. Giving the participants the time to accomplish their tasks without limits could have an effect on the time measurement. If a subject was stubborn enough to spend a long time on one of the constraints, this could affect the average time required to define the constraint. However, limiting the time will not allow some subjects to continue the definition process and the constraints will be considered defined wrongly which may leave a doubt about the correctness measures. In addition, limiting the time could increase the complexity of the statistical analysis.

Another threat to this experiment is the individual differences between the constraints themselves. Some of these constraints were implemented with the expectation that they would be expressed negatively, others positively and others using a negative example with action. This may open a question about the effect of redistributing these factors over the constraints in a different way than used. Dependence on individual differences of the constraints could have been reduced by increasing the constraint set for each task. However, this was not feasible given the several variables to be counterbalanced - the tool implementation and the natural language the constraints are written in, and constraint presentation order. This could be solved by using more subjects but this was not possible in this experiment because of the time, resources and difficulty of finding subjects. However, this introduces an idea for future research that may study individual differences between the constraints; however, this question is not part of the aim of this study.

The subjects' native language is also a threat to validity as not all the subjects are native English speakers. However, all the participants are studying or doing research in English at Glasgow University and thus considered sufficiently fluent in English to understand the constraints presented to them.

With respect to external threats, using only one diagram type, STD, may limit the generality of the results. The decision of using this diagram type has been justified before. Additionally, time and resource limitations prevented investigating more diagram types.

6.3.7 Related Work

Since PBE has not been applied in a meta-CASE tool before, the ideas in this research and DECS features will be compared to other relevant PBE systems. Many previously developed PBE systems support both positive and negative examples. Heffernan (2003) proposed the use of positive and negative examples to enhance an intelligent tutoring system. Myers (1993) observed that Peridot depends mainly on positive examples; however, some constraints required being expressed using negative examples. Peridot is the closest system to DECS, although they are different in terms of application domain, as Peridot infers graphical user interface constraints. This difference in constraint nature and application context most likely affected the

need for different example polarity since it seems that there was no problem in just using the positive examples only in Peridot as very few of its constraints depend on negative examples.

Unlike DECS, all the reviewed PBE systems, apart from Peridot, that report using positive and negative examples depend on those two example types working together to refine the specification. In Gamut (McDaniel & Myers, 1999), negative examples are used to exclude behaviour from a generalised, positive one. MetaMouse (Myers, McDaniel, & Wolber, 2000) uses implicit negative examples to refine behaviour through conditional branches in the code. Usually such systems are very sensitive to user errors or slightly malformed examples. The InferenceBear (aka Grizzly Bear) system users found using positive and negative feature difficult (Myers, McDaniel, & Wolber, 2000).

Hudson & Hsi (1993) criticised using different example polarities to refine generated code and system behaviour because this makes it very demanding for the user to define the behaviour. Instead, they recommend involving the user in selecting the required behaviour from alternatives. DECS follows this approach in part. DECS depends on positive and negative examples but uses each as a standalone example, not as refinement for the other. This offers the usefulness of using different polarities to increase the expressiveness of the examples while keeping it relatively easy to use by not involving the user in refinement issues. None of the systems above used positive and negative examples as two separate types, each of which is used to provide a completely separate example. As far as the author is aware, applying negative and positive examples as a “more natural” way of expression has not been tried before nor has there been a previous empirical study to explore this issue¹⁰.

¹⁰ There is a suggestion in (Myers, McDaniel, & Wolber (2000) that a small experiment was conducted on the InferenceBear/Grizzly Bear system but this has not been published.

6.4 Adding and Customising Rules (STUDY FOUR)

6.4.1 Introduction

The CSBE technique has proven its superiority over the form-filling technique as has been shown in Study One (6.1). Study Three (6.3) has also shown that providing the opportunity of expressing the constraint using different polarity (multi-polarity) examples enhances CSBE. As concluded from the last section, allowing the user to express the constraint using either positive or negative examples gives a richer set of alternative examples of which some may be perceived to be more natural than others.

DECS depends on a set of rules to infer the intended constraints using examples. Consequently, the inference engine ability to infer constraints depends on its knowledge. DECS' ability to interpret examples is limited to the fixed set of rules implemented in the DECS inference engine. Consequently, if the inference engine does not have the required knowledge (rules) to interpret an example, DECS will not be able to infer the required constraint.

DECS knowledge can be augmented by adding rules in the form of strings to the inference engine. This must be augmented by providing associated Java classes, if not available, to be used at runtime. Consequently, a programmer is required to manipulate the code to handle the knowledge and rule augmentation problem in DECS. Clearly, it would be desirable to be able to offer a way to add rules to DECS at runtime and avoiding direct code manipulation.

The rules augmentation problem extends beyond adding rules that the inference engine is missing; there is also the challenge of rule customisation. This describes the case where DECS has the knowledge to infer the required constraint, but it is difficult for the user to think of the required example to express the constraint. That is, there may be an alternative potential example that is better (closer to the user mind) suited to the way a particular user thinks about the constraint and its expression. In other words, the problem can be viewed as forcing CSBE users to adapt themselves to it rather than CSBE adapting itself to its users.

The simplest example of this problem is that a user prefers to express a constraint using a negative example while the inference engine is implemented to infer the required constraint using a positive example. This creates several problems starting with the extra effort and time that will be consumed in trying two different ways of expressing the constraint. This part of the problem will not be limited to the current constraint as every time the constraint or a similar one is required, the user will try to express it according to their way of thinking about the example. Consequently, the user will spend the same time and effort trying the different alternatives each time they try to express the same or a similar constraint. One solution for this problem is that the user learns how to express the constraints using the examples that the system understands. However, this imposes a burden on the user, requiring them to adjust their way of thinking to suit the system and to remember the particular conceptualisation of the system. This waives the advantages of facilitating constraint specification and the intuitiveness of expressing the constraints, and provides the example that the system prefers to express the constraint not the one that the user prefers. This introduces the second problem of the need to tailor or customise the inference according to the user way of thinking not the developer one (the system).

This problem was noticed several times during the previous two empirical studies. The participants often tried to express the constraints using examples other than those thought about by the author and implemented in DECS. In other words, the users did not share with the author, and thus with DECS, the same notion of how to express the constraints via examples. This is not unexpected as different people think in different ways. Although the problem might seem to be a positive and negative example problem, it is believed that users may vary in expressing a constraint even with the same polarity. Thus the problem is one of tailoring which includes the polarity preference problem as a subset.

Chapter 4 introduced the technical details for the solution of the above two problems (adding knowledge and customising the tool) through a learning technique. In this technique the user teaches DECS how to define a constraint using an example that is more natural to the user. The learning technique depends on the user introducing an example and using the wizard to specify the constraint. DECS watches

and learns how to define the constraint using the example. The next time the user introduces the same or a similar example, DECS is able to infer the required constraint. This solves both problems, adding and customising rules, introduced above. This section starts by recalling the solution and some of the technical details. The section also presents and describes an empirical study conducted to evaluate the feasibility and desirability of the implemented learning technique. The experiment is designed and conducted to answer the question:

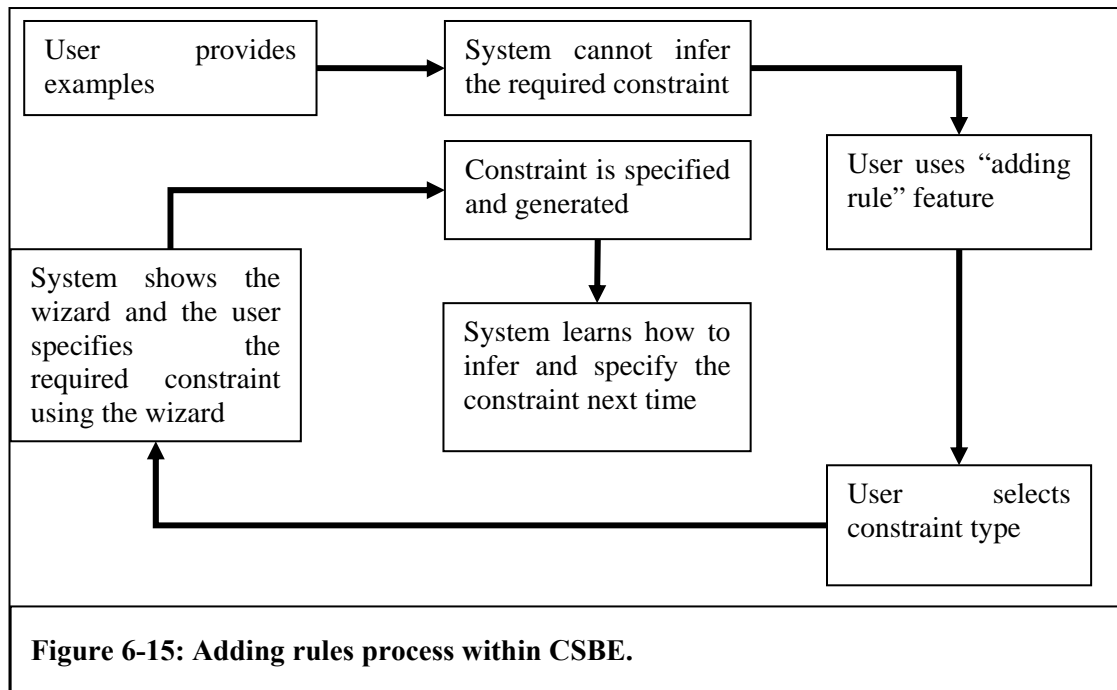
Does implementing and using the learning technique influence the performance of CSBE technique?

This study tests the claim stating that *it is possible to increase the performance of CSBE by implementing a learning technique for augmenting and customising the knowledge base of the system.*

The experimental design is described and the results presented and discussed. The section draws some conclusions from this experiment and reviews some related literature.

6.4.2 Proposed Solution

The proposed idea to solve the above two problems is by allowing the user to introduce an example that expresses the constraint (according to the user). Later on, the user specifies the required constraint using the wizard. This is considered as a system learning technique and also it has been called an “adding rules” features throughout this research because it involves augmenting the inference engine with rules. This technique is illustrated in Figure 6-15, which is considered as a subset of Figure 4-1 (recall from Page 97) with more focusing on the learning part of the model. Figure 6-15 is explained in the following scenario:



To recall from Chapter 4, adding rule feature depends on that the user introduces an example that s/he believes expresses the required constraint. The system infers according to its knowledge. The required (intended) constraint does not appear in the constraint list shown by the system. The user presses a button, labelled “Add Rule”, and selects the required constraint type (graph, vertex, or edge constraint) which activates the appropriate wizard. The user specifies the required constraint with a suitable description in natural language and saves it. This means that the user has specified the required constraint using the form-filling technique at the same time s/he is adding a rule. In other words, the rule addition process is a dual purpose task, adding the rule and specifying the required constraint at the same time. The system also saves the introduced example as a Java object and attaches it to the specified constraint. The next time the user introduces the same example, the system searches both the original rules in its knowledge base and the added examples. It finds a match from the saved added examples (because the new and the old examples are the same) and includes the constraint description into the inferred constraint list. When the user selects the constraint and confirms the selection, the system uses the previously specified constraint (constraint specified using the wizard) to specify the new constraint. The new constraint is generated and saved.

The following section documents an empirical study that was conducted to evaluate the feasibility and desirability of the implemented learning technique.

6.4.3 The Feasibility and Desirability of Adding Rule and Customisation Feature

6.4.3.1 Aim, Variables and Hypothesis

The aim of this experiment is to evaluate the learning technique implemented in the DECS inference manager. The experiment studies and evaluates the following:

- The feasibility and desirability of enabling the system to learn to infer and define new constraints that did not exist in its inference engine knowledge.
- The feasibility, and affect on constraint specification performance, of enabling a user to employ examples that they believe are more natural for expressing a required constraint, by adding a rule addition capability to DECS.
- The experiment also studies the feasibility, desirability and the performance effect of the generalisation feature implemented in the learning technique.

6.4.3.2 The independent variables:

To set up the independent variables of the experiment, one DECS implementation has been used. This DECS version was intentionally implemented to be unable to infer some constraints and to infer some others using complex examples only (this will be discussed in detail later on). The users were encouraged to teach the system to recognise constraints from new examples that they introduced, either to add new example-constraint rules they considered missing from the system or to customise rules that they considered difficult to express.

According to the above description the independent variables of the experiment are the two conditions:

- A DECS version that does not have the required knowledge in its inference manager to infer all the required constraints and can infer some other required constraints but using difficult examples. This DECS version required to be taught using “adding rule feature” how to infer and specify the required constraints. Accordingly, this variable is considered as “before the system learned” condition.

- The same DECS version described in the above bullet point after has been taught using “adding rule feature” how to infer and specify the required constraints using the user preferred examples. Accordingly, this variable is considered as “after the system learned” condition.

6.4.3.3 *The dependent variables*

For the purpose of achieving the aims of the study, the following potential dependent variable measurements were tested before and after the system has been taught how to specify constraints.

- The *effectiveness* of the system in terms of the resulting constraint specification **correctness**. This measures the number of constraints defined correctly (out of 6) in the two conditions (tasks), before the user teaches the system and after the user teaches the system.
- The *effectiveness* in terms of the frequency that the “adding rule” feature has been used. Each time the user is required to use the adding rule feature, the constraint is considered wrong. This dependent variable measures the **number of instances of rule addition** during the two tasks, before and after the user teaches the system.
- The *efficiency* in terms of the **time required** for accomplishing the constraint specification task. This is a measurement of the time that the user requires to finish each of the two tasks.
- The **user satisfaction** with the learning technique. This elicits if the users think that the rule addition feature is useful and whether or not they prefer using it.

6.4.3.4 *The Hypothesis:*

The null hypothesis of this experiment states that,

H0: *there is no difference between “before the system learned” and “after the system learned” how to specify constraints regarding the effectiveness, efficiency and user satisfaction.*

The alternative hypothesis states that,

H1: *performance is improved after the system has learned how to specify constraints with respect to the effectiveness, efficiency and user satisfaction.*

6.4.3.5 Data Collection and Tasks

The study required users to carry out a set of supplied constraint definition tasks. A State Transition Diagram (hereafter STD) was used as the context for this experiment. STD was selected because it is commonly used in the software design process and all the participants were familiar with it. This diagram also contains by default all the general constraints that may appear in most other diagram types. However, the constraints that were used were modified to allow evaluation of the generalisation feature as part of the learning technique. The constraint modifications included customising different properties such the vertex and edge types, colours, and the vertex and edge upper bound number.

For the purpose of conducting the experiment, 16 participants were selected from Computing Science and Software Engineering postgraduate students at Glasgow University. The participants were invited by an email that contains all the information about the experiment and the requirements of the participants. The participants were required to be familiar with STD and they must not have participated before in any previous experiment for this research. The last requirement was set out because it was noticed during the pilot studies that participants who used the system before in previous experiments, especially the last experiment, were expecting the system to be intelligent and to infer the required constraints from their examples. Consequently, there was a need for participants with no background expectations about the system.

Two constraint lists with six constraints each were created (Appendix F). Both lists contained similar, but not identical, constraints. The constraints were selected from different categories resulting in the following set:

- one constraint related to the cardinality of vertices,
- two label-related constraints,
- one unique visual representation constraint,
- one connection between vertices and path related constraint, and

- one lower bound number connection constraint.

The constraints were selected carefully based on different criteria. The first is that the constraints must have different properties, such as colour and vertex and edge types, so that the generalisation feature can be tested. They also were chosen to be comprehensive so they evaluate different generalisation possibilities including the effect of the limitation discussed above.

One of the criteria in choosing the participating constraints was the results from Study Three. Two label-related constraints were selected because they proved to be easy to express in Study Three (Figure 6-12 – constraint 3) (easy in this context means that the constraint scored higher correctness and lower specification time than other constraints). These two constraints were placed in the two lists, one in each list. These constraints were implemented so that they could be inferred easily (according to the previous empirical studies) without requiring examples to be added or customised. Their existence was intended to give the user the confidence that the tool is working correctly and there is an inference engine that can infer some constraints. This, hopefully, reduced the doubt of the user that the tool cannot infer any constraint and all the constraint must be taught to the system which probably would reduce the motivation to participate in the experiment.

The other constraints were chosen, based on results from the Study One and Study Three, so that their difficulty to be expressed ranged from easy, such as the cardinality constraint, to difficult, such as the path related one (difficult in this context means that the constraint scored lower correctness and higher specification time than other constraints). Table 6-9 shows one of the two lists used, with a description of each constraint used and the purpose for its use. The table shows the constraints of list 1 as a representative of the constraint categories used. It shows for each constraint if it is considered as difficult or easy. Additionally, it shows the property in the constraint that requires to be generalised. This property will be the difference between the first list (used in the task before the system taught or the task *at* teaching time) and the second list (used in the task after the system been taught in the first task).

Table 6-9: Justification of the reasons for using each constraint in one of the lists used in the study.

Constraint (first list)	Constraint (second list)	Diff	Easy	Required Generalisation
1: At most 3 StartState (s) are allowed in the diagram.	At most 4 EndState (s) are allowed in the diagram.		X	Vertex type. Cardinality number.
2: NonTerminalState (s) must have unique labels.	NonTerminalState (s) must have labels.		X	No generalisation.
3: StartState must have unique visual representation in any given diagram.	EndState must have unique visual representation in any given diagram.	X		Vertex type.
4: Any Transition edge label must start with the substring “out”.	Any Transition edge label must start with the substring “_in”.	X		The label string.
5: There must be a path between Red StartState and the Green NonTerminalState in the diagram.	There must be a path between the Yellow NonTerminalState and Blue EndState in the diagram.	X		Starting vertex type. Ending vertex type. Vertices colours.
6: There must be at least 1 Red Transition edge connecting StartState (source) with NonTerminalState or EndState (target)	There must be at least 1 Green Transition edge connecting NonTerminalState (source) with NonTerminalState or EndState (target).		X	Starting vertex type. Transition edge colour.

As explained above, each constraint list has six constraints. Three of these constraints (constraints number 2, 3, and 4 in Table 6-9) can be expressed using the tool without requiring any additional inference rules; however, two of these three constraints (3, and 4) are expressed using putatively difficult examples based on observations from Study One and Study Three. The third constraint, the label-related one (constraint number 2), can be expressed using an easy example, as discussed above in this section. These three constraints were used to evaluate the feasibility and desirability of the tool for customising the examples that express the constraints. This is because the constraints can be expressed using the tool but the user may not be able to find the implemented example required to infer the constraint. In this case, the user

would be required to teach the system the example they prefer to express the constraint.

The other three constraints cannot be expressed using the tool, as the rules that infer them were removed from the DECS inference engine. These constraints evaluate the feasibility and desirability of teaching the system entirely new example-constraint inference rules. During task one, the user was instructed to try to specify the constraints and when s/he cannot, to use the rule addition feature to teach the system how to infer and specify the required constraints. This is called the first task. Later on, the user is asked to specify the constraints in the second constraint list, which contains similar but not identical constraints as the first one. This is called the second task.

Table 6-10: Constraints used in the study and the ability of the system to infer them before using the adding rule feature (teaching the system).

Constraint Num	Can be inferred without learning?	The purpose of use in the study.
1	No	Requires using adding rule feature. Study the generalisation of vertex type and cardinality.
2	Yes	Easy to be expressed to give the user the confidence that the tool can infer something and not all the constraints required to be taught.
3	Yes	Assumed to be customised. Study the generalisation of vertex type. A possible example could contain action.
4	Yes	Assumed to be customised. Study the generalisation of label strings.
5	No	Requires using adding rule feature. Study the generalisation of more than one property together (vertex type and colour). A possible example (if negative) can show the limitation of the generalisation feature.
6	No	Required to be added. Study the generalisation of more than one property together (vertex type and edge colour).

The second task is conducted using the same tool that is used for the first task. This means that the user teaches the tool in the first task and uses this taught tool in the second task. As is shown in Table 6-9, both constraint lists contain similar but not identical constraints. The differences between the constraints in the two lists relate to the following features: the number of vertices in the graph, the vertex type, the edge type, the vertex colour, the edge colour and the edge label. These features are used to study the example generalisation.

Table 6-10 shows the constraints used in the study, including their identification number, whether or not the constraint can be inferred by the tool without (or before) using adding rule feature, and the purpose of its use in the study.

6.4.3.6 Experimental Design and Execution

As described above, two constraint lists were designed to contain similar, but not identical, constraints. One of these lists was used as the list in the constraint definition task before teaching the tool while the second list was used as the list in the constraint definition task after teaching the tool. In each list, the constraints were written into two natural language forms, viz., positive and negative, in English. This results in 4 constraints lists (differences in constraints themselves and differences in natural language polarity) (see Appendix F). These 4 constraint lists were counterbalanced by alternating the constraint list the user starts with in task one and alternating the language within the constraint list. These lists were assigned to the different users before starting the experiment. Since 16 users participated, each four users used the same alternative combination. This guaranteed the counterbalancing of different conditions that might affect the experiment. The distribution of the lists and assignment to different users are summarised in Table 6-11 which shows the lists used in the form of (user number (first list used / second list used)).

Table 6-11: Assignment of lists to users.

16 Users			
Positive Language		Negative Language	
User 1 (list 1/list 2)	User 2 (list 1/list 2)	User 9 (list 3/list 4)	User 10 (list 3/list 4)
User 3 (list 1/list 2)	User 4 (list 1/list 2)	User 11 (list 3/list 4)	User 12 (list 3/list 4)
User 5 (list 2 / list 1)	User 6 (list 2 / list 1)	User 13 (list 4/list 3)	User 14 (list 4/list 3)
User 7 (list 2/list 1)	User 8 (list 2/list 1)	User 15 (list 4/list 3)	User 16 (list 4/list 3)

For this evaluation a within-subject design was adopted. For each participant two tasks each with a constraint list composed of 6 constraints were used. Each task is to specify the constraints in the list using the CSBE technique. However, the first task involved teaching the system how to specify the constraints while the second involved using the taught constraints to evaluate the ability of the system to learn from the first task. If the constraint could not be inferred for any reason in task two, users were instructed to teach it to the system using the rule addition feature as in task one.

Each participant was trained for about 20 minutes. The training included explaining the different ways of expressing constraints and the polarities of that expression. Users were also allowed to “play” freely with the system. This aims to familiarise the user with the system to be able to handle it without problems in dragging and dropping the vertices, connecting them using edges, and changing the different properties. However, the training did not use any constraint list. This was to allow the user to provide examples according to his/her thinking instead of being biased by the examples they trained on. Participants were asked to fill out a number of different questionnaires at different stages of the experiment and a short interview

was conducted with them at the end. All the tasks were recorded by screen capture to be used later for data extraction and analysis.

When the user performs the first task, s/he is asked to specify the 6 provided constraints. Since it is not possible to specify, at least, 3 of them because the system cannot infer them, the user needs to specify the constraints using the wizard through “adding rule” feature. During this process, the system learns from the user how to express and specify the constraints. However, because the learning process involves using the wizard and using the wizard is an experiment by itself that needs user training, this part (using the wizard) was done by the researcher. This was conducted as follows:

The user reads the constraint from the list and tries to express it using a convenient example (according to him / her) and checks the inferences. If the user tries to express the constraint using the preferred example but the system does not infer the required constraint the user asks the researcher to define the constraint for him/her using the wizard. The researcher specifies the constraint using the wizard with detailed explanation for every action in the wizard to keep the user involved and attracted to the process. This solves the problem of the need to train the user on using the wizard and avoids any mistakes that the user may do if they used the wizard themselves to specify the constraint. If there is a mistake in defining the constraint using the wizard, this means that the system has been taught wrongly and will not be able to infer or generalise correctly. In addition, using wizard in constraint specification has been evaluated before in this research and no need to be evaluated again in this experiment.

To acquire the user perceptions and opinions, each user was asked to fill two types of questionnaires, post task (after each task) and post experiment (after the whole experiment). These are in total 3 questionnaires. For the all the questions in the post task questionnaires a Likert 5-point scale was used and some of these were inverted to reduce bias. Similarly the questions in the post experiment questionnaire with additional semi-structured interviews with the users to elicit the opinions about the adding rules feature.

6.4.4 Results

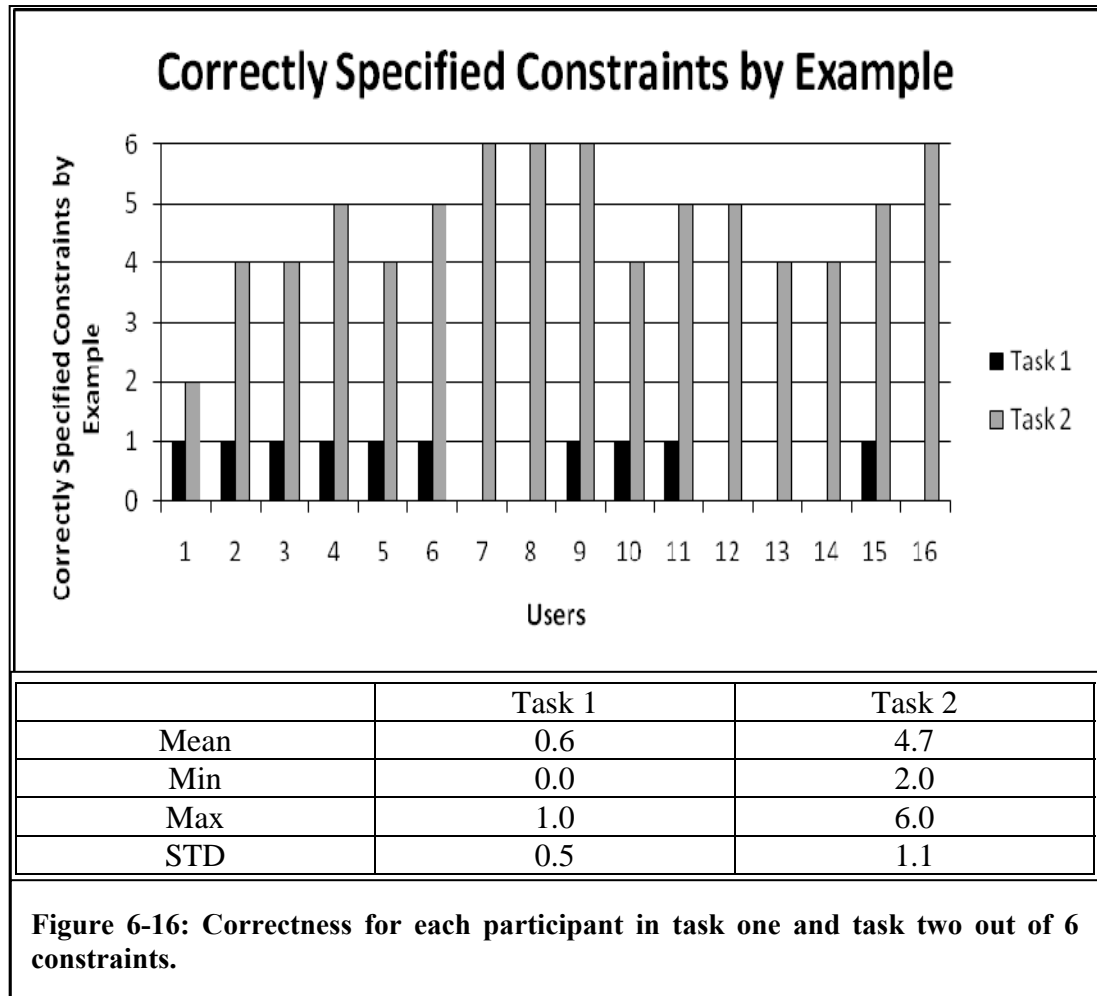
All the 16 participants were familiar with the STD used in the evaluation. Most of the participants indicated that they were familiar to some degree with the constraint definition task with an average of 3.06 for the first task and 4.19 for the second task on a scale of 5. Each experiment took between 60 and 90 minutes including training and questionnaire filling time. The results for the users' attempts were analysed with respect to the above mentioned hypothesis. No normality assumptions were made about the collected data. Consequently, the nonparametric Wilcoxon Signed Ranked test was used to analyse correctness, the number of constraints requiring the rule addition feature and time required to accomplish the task in addition to the satisfaction results. The comparison between the two techniques regarding these measurements is presented in the following sections.

6.4.4.1 Correctness

6.4.4.1.1 Correctness for each participant

The number of correctly specified constraints (out of 6 constraints) for each participant in both tasks was gathered from the recorded screen capture videos. The constraint was considered to be specified correctly if and only if the user provided an example that leads the system to infer the constraint. Based on this, a constraint that the system could not infer and, consequently, that required that the user teach it to the system using the wizard, was not considered correct. It can be argued here that half of the constraints cannot be inferred by the system in task one which means it is unfair to compare task one with task two. However, this is part of the aim of the experiment as it evaluates the effect of the learning technique on the ability to specify constraints via CSBE.

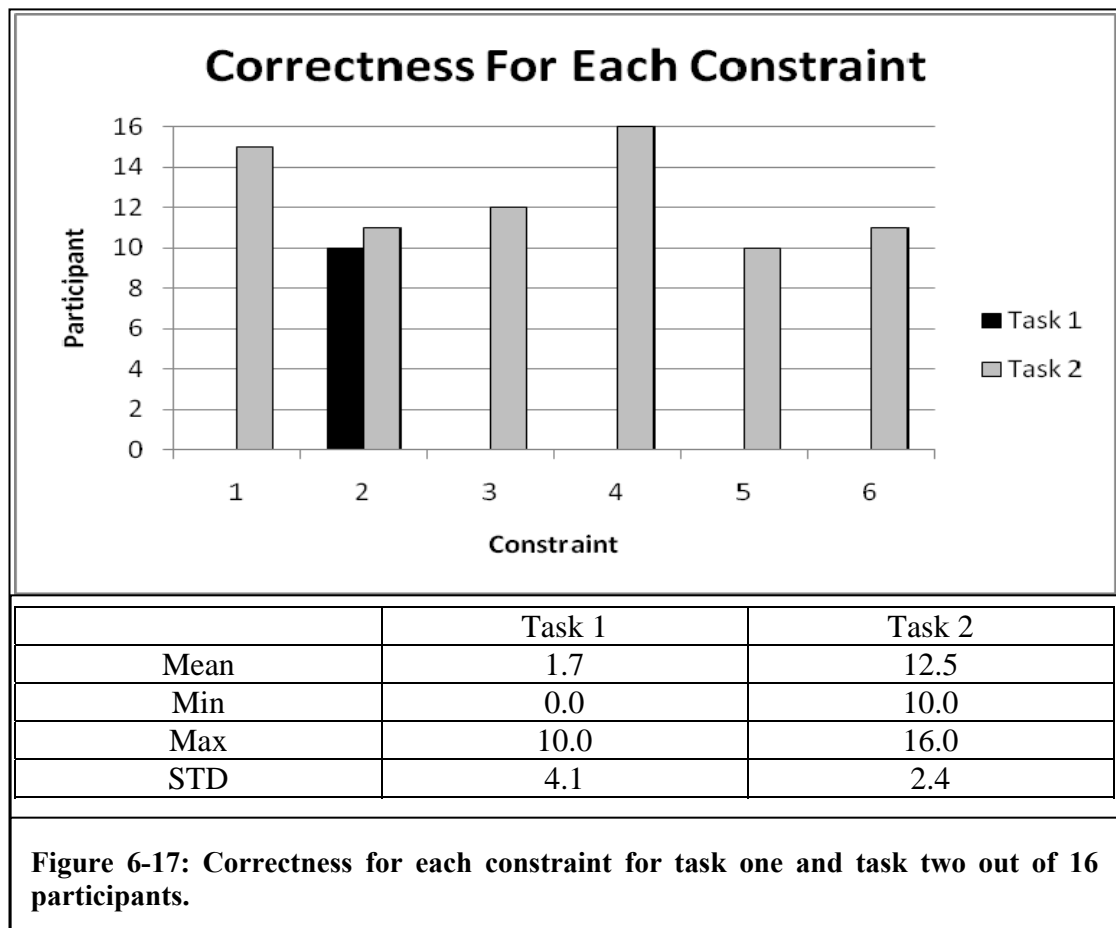
All the constraints were attempted by all the users. Figure 6-16 shows the number of correctly specified constraints in task one and task two for each participant. A significantly higher number of constraints were specified correctly in task two than in task one.



All of the participants defined a higher number of constraints correctly in the second task than the first one (Figure 6-16). In task two, 4 participants (25%) defined all the constraint correctly (no need to use the rule addition feature). In task one, 6 participants (37.5%) specified all the constraints incorrectly (i.e., needed to use rule addition for all the constraints) and of the rest, 10 participants (62.5%), defined only 1 constraint correctly in task one. In all cases the correctly defined constraint in task one was the label-related constraint, which was known from the previous experiments as easy to express (Section 6.4.3.5). The lowest number of correctly defined constraints in task two was 2 constraints. Analysis shows that there is a highly significant difference between both tasks regarding correctness in constraint definition ($p < 0.001$).

6.4.4.1.2 Correctness for each constraint

A constraint is considered correct if it correctly captured the constraint description given in the task and the user was able to specify it without use of the rule addition feature. Task two outperforms task one regarding the average number of users defining constraints correctly (correctness per constraint) with an average of 12.5 participants defining each constraint correctly compared to 1.67 in task one. In task one, constraint number 2 (the label related one) was the only correctly defined constraint, due to the need to use rule addition for the others. For constraint number 2, only 6 users needed to use rule addition while the other 10 users defined it correctly without rule addition. By contrast, in task two, the lowest number of correctly defined constraints was 10 participants in constraint number 5, the path related constraint.



The rest of the constraints were defined by a higher number of participants with the highest number (16 participants, i.e., all of the participants) for constraint number 4, also a label related constraint. The next highest correctness result was for

constraint number 1, the vertex cardinality constraint, with (15) participants. Data analysis showed a significant difference between the two tasks regarding the number of participants that defined the constraints correctly ($p = 0.028$, $Z = -2.201$).

Figure 6-17 shows this data which can be considered as the same data shown above Figure 6-16 but with different presentation and perspective.

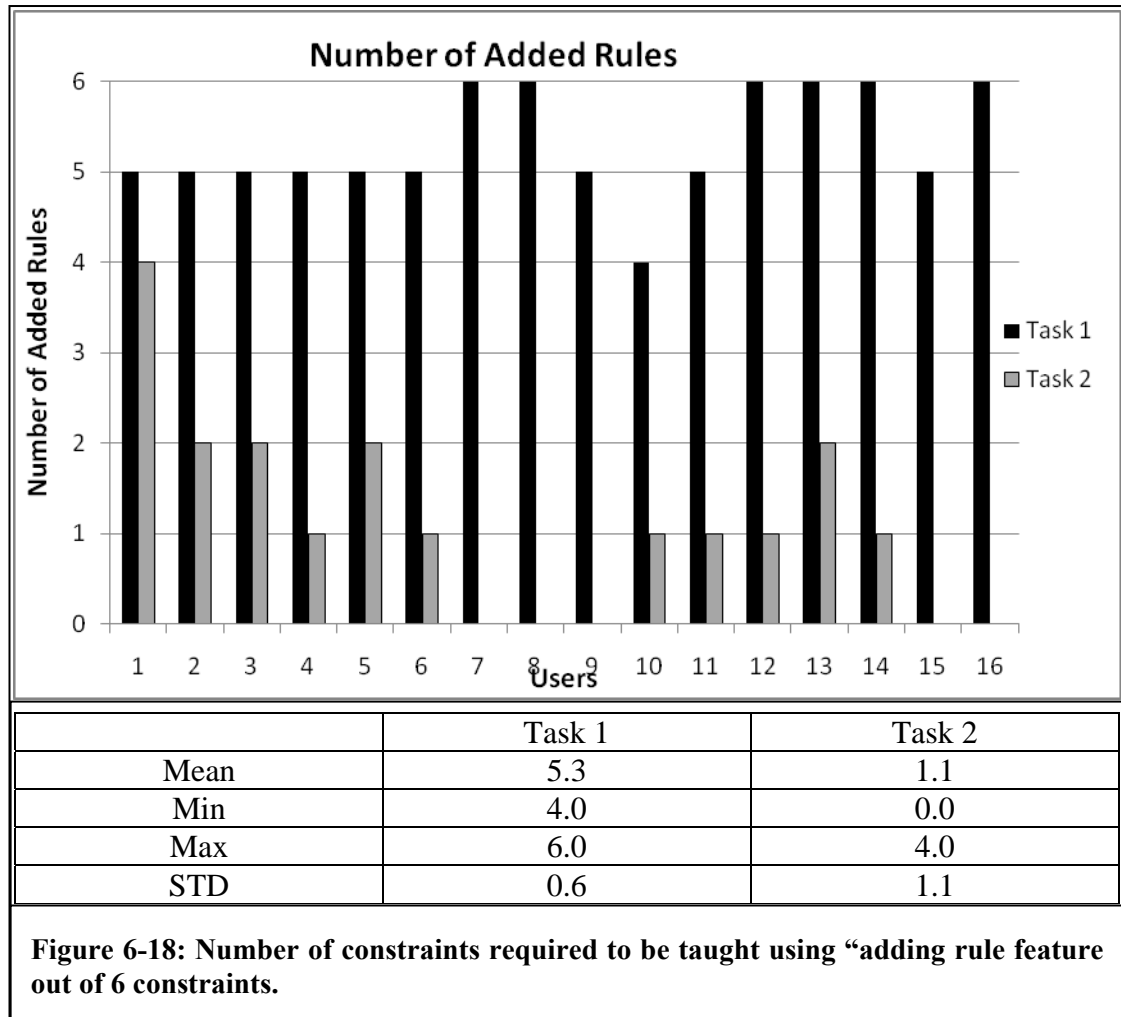
6.4.4.2 Number of constraints required to be taught (added)

6.4.4.2.1 Number of constraints required to be taught for each participant

The number of times the rule addition feature required to be used for specifying and teaching the system inferring and specifying the constraints in both tasks was gathered from the recorded screen capture videos. The constraint was considered to be taught if and only if the user asked the researcher to use the rule addition feature to specify the constraint. Any constraint that has been taught was counted.

Figure 6-18 shows the number of constraints required to be taught using the rule addition feature in task one and task two for each participant. A significantly higher number of constraints were specified and taught using the wizard in task one than in task two.

All of the participants required to use the “adding rule” feature for teaching the system more frequently in task one than task two (Figure 6-18). The average numbers of constraints used “adding rule” feature were 5.3 constraints and 1.13 constraints for both task one and task two respectively. 6 participants (37%) required “adding rule” feature to specify all the constraint in task one. Only 11 participants (68.8%) needed the “adding rule” feature in task two, 6 of them used it for only 1 constraint and the highest requirement was to specify 4 constraints of the 6. 5 participants have not used the feature at all in task two. Analysis shows that there is a highly significant difference between both tasks regarding frequency of teaching the system the constraint specification ($p < 0.001$).

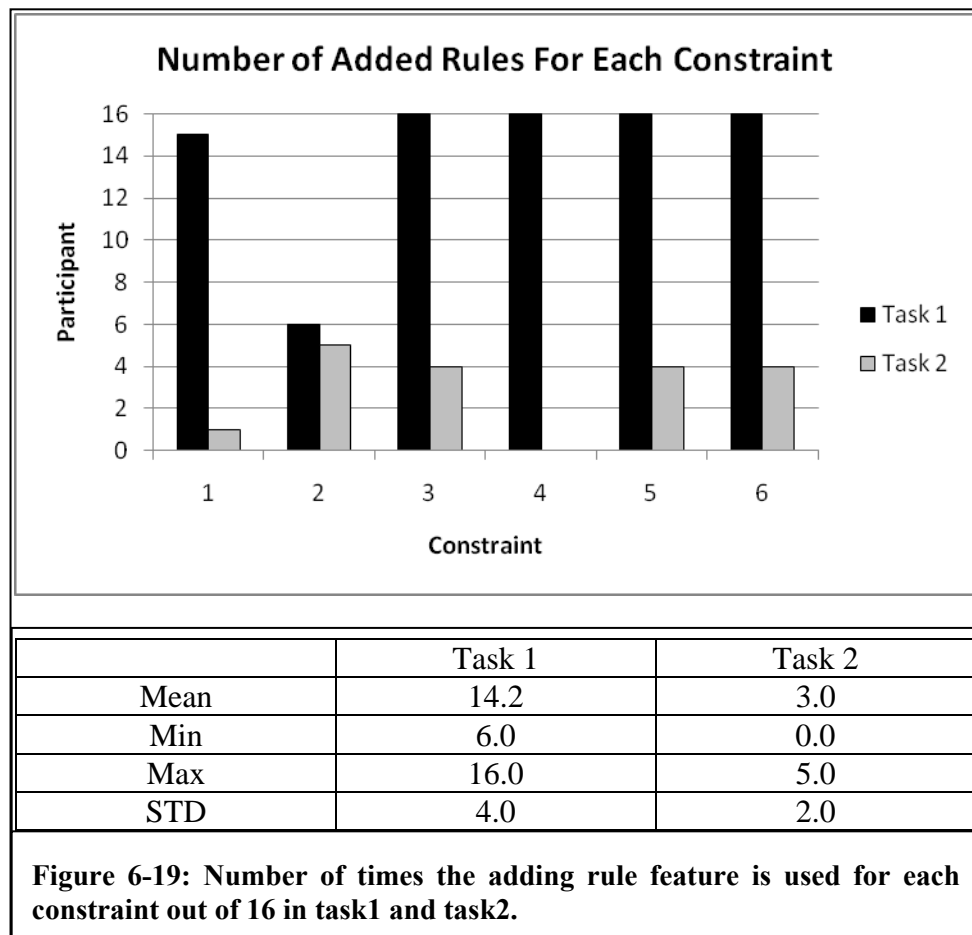


6.4.4.2.2 Number of constraints required to be taught for each constraint

The number of constraints required to be taught or the number of rules to be added for each constraint was calculated (Figure 6-19). This shows the same data as Figure 6-17 and Figure 6-18 but from a different perspective. Figure 6-19 is considered as a reflection of Figure 6-17 since any constraint that required to be added (Figure 6-19) is considered as an incorrect constraint (Figure 6-17). Figure 6-19 is introduced for more clarification, although its data can be inferred from (Figure 6-17).

All the users required to add rules for the last 4 constraints in task one. Only one user defined the first constraint correctly out of the 16 users as 15 of them (93.7%) required to add it. For constraint number 2, discussed above, only 6 participants required to add it (37.5%) in task one as it was designed to be easy. In task two, the most frequently added constraint was the label related constraint number 2; 5 participants added a new rule to infer it, as it had not been added or customised in

task one. Note that adding this constraint in task one will not help at all in task two as the constraint is different although it is still in the category of label related constraints and it has been implemented in task two to be easy to express. 3 constraints were also added by 4 participants each, one constraint was added by one participant (constraint number 1), and one constraint was defined correctly without the requirement to be added again by any participant in task two.



Statistical analysis shows a significant difference between task one and task two with ($p = 0.026$, $Z = -2.226$). This significant difference is expected because the of significance found in (Figure 6-17 and Figure 6-18).

For extra clarification for the correctness and added rules results for each constraint, the following table shows the constraints that are required to be added represented by dark shaded cells. Table 6-12 represents task one while Table 6-13 represents task two.

Table 6-12: The constraints that required the use of rule addition feature in task one (shaded = added).

Constraint #	Users															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1																
2																
3																
4																
5																
6																

Table 6-13: The constraints that required the use of rule addition feature in task two (shaded = added).

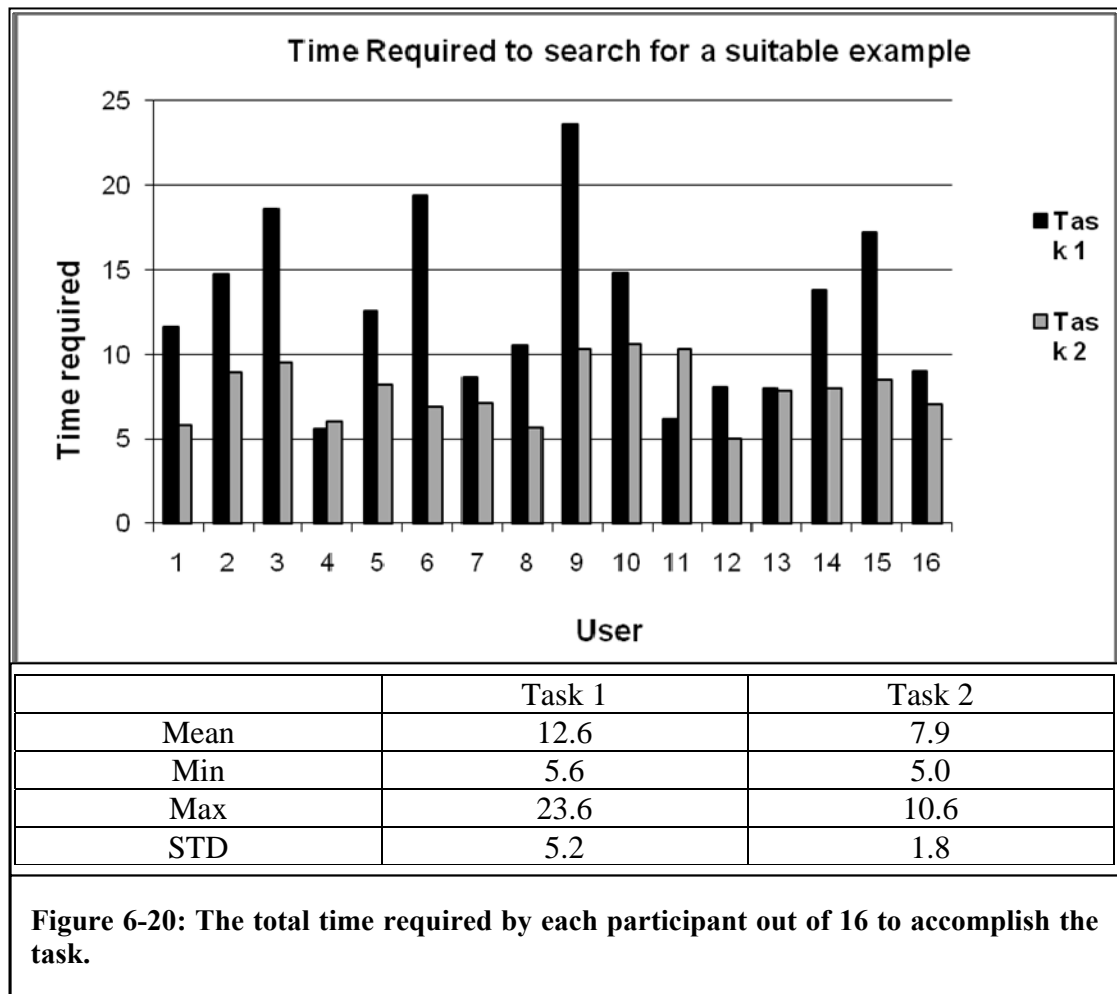
Constraint #	Users															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1																
2																
3																
4																
5																
6																

6.4.4.3 Time

6.4.4.3.1 Sum of time required by each participant to complete the tasks

The time required to accomplish the constraint definition task for each participant in both tasks was gathered from the recorded screen videos. The time for each participant was rounded by a factor of 15 seconds. The time to accomplish the task for each participant was calculated by summing the time required to define each constraint in each task. The time was counted for each constraint starting from the point when the currently finished example is deleted from the editor and the editor is cleared and ready for another constraint. The end of the time count (the counter is stopped) is when either the user decided to use the “adding rule” feature by pressing the “add rule” button or the user finished defining a constraint using CSBE when the system succeeded in inferring the constraint and the user selected the correct inference. The time was counted whether or not the constraint was defined correctly. Figure 6-20 shows the time required for each user to accomplish both tasks in the

STD. The diagram shows that task two has better results (a shorter time) than task one and analysis shows that the difference between the two tasks is significant.



14 out of 16 (87.5%) participants required more time to specify constraints in task one than in task two. The time averages required to accomplish the tasks for each user are 12.6 minutes and 7.9 minutes for task one and task two, respectively. The highest time required was 23.6 minutes and 10.6 minutes for task one and task two respectively. Analysis shows that there is a highly significant difference between both tasks regarding frequency of teaching the system the constraint specification ($p = 0.002$).

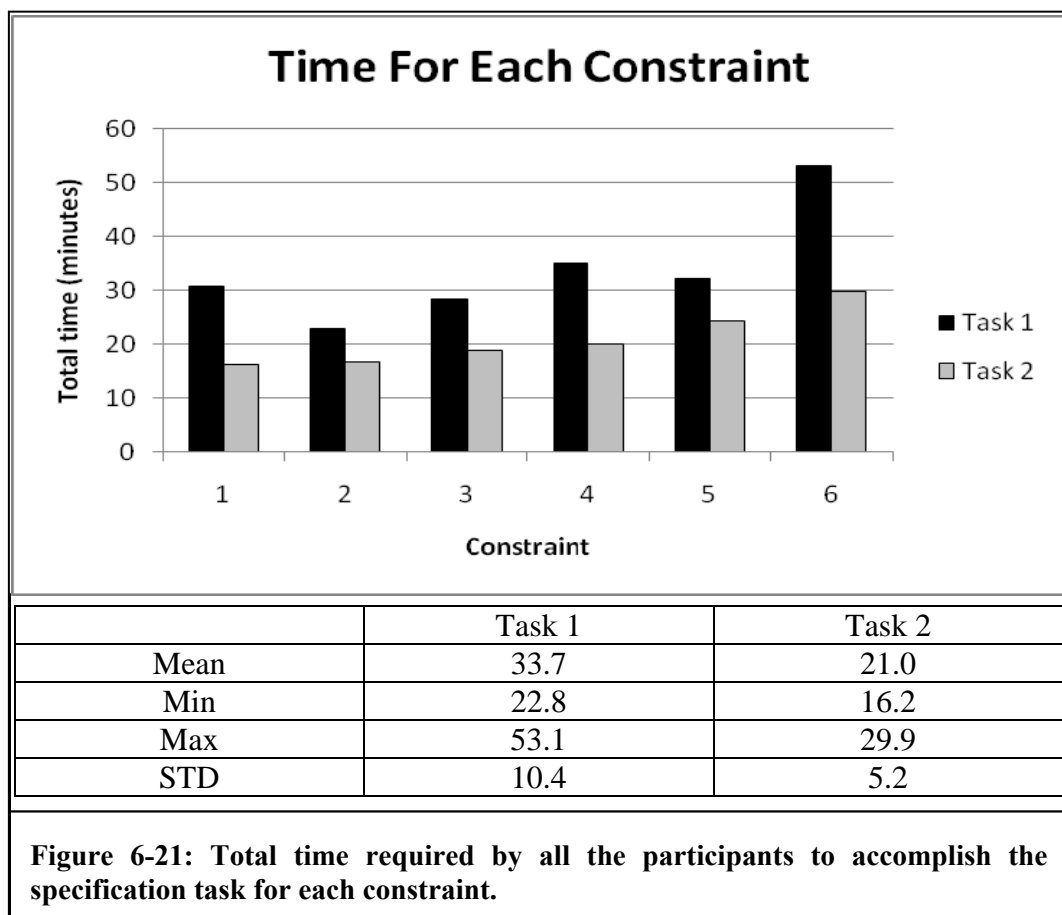
6.4.4.3.2 Sum of time required by all the participants for each constraint

The total time required by all the participants to accomplish each constraint definition was calculated and presented (Figure 6-21). Constraint number 2 required the shortest time among all the constraints in task one. The rest of the constraints fall

in the range 28 to 35 minutes with the exception of constraint number 6 with a total time of 53 minutes. The average time required for each constraint in task one was 33.68 minutes.

In task two, the shortest required time was for constraint number 1 with a time of 16.2 minutes, followed by constraint number 2 again with a slight difference, 16.7 minutes. The rest of the constraints are close to each other, ranging between 18.9 and 29.9 minutes. The average time required for each constraint in task two was 21.0 minutes.

Statistical analysis shows a significant difference between the two tasks ($p = 0.028$, $Z = -2.201$). The significance was expected because of the existence of significance between the two tasks in the case of time required to accomplish each task by each participant, presented above.



6.4.4.4 User Perception

In order to provide further validation of the hypothesis, post task and post experiment questionnaires that participants filled out were analysed.

In post task questionnaires (filled in after each technique), participants' opinions about the constraint definition tasks were investigated. There was a need to discover the participant experience while interacting with the system before and after the learning technique is used. Table 6-14 shows the most significant questions and the average answer scores.

Table 6-14: Average answers of the post-task questionnaire. (shaded = better (higher user satisfaction), bold and underline = significant difference).

Questions	Task one (out of 5)	Task two (out of 5)
How mentally demanding was the task using this technique?	3.25	<u>2.06</u>
How temporal demanding was the task?	2	<u>1.44</u>
How successful were you in accomplishing what you were asked to do?	2.44	<u>2</u>
How uncertain, discouraged, irritated, stressed, and annoyed were you?	2.63	<u>1.88</u>
The constraint definition task was easy using this tool.	3	<u>3.75</u>
While I was working, I felt that I needed help from an expert.	3.19	<u>2.25</u>
I achieved the required constraint with my first attempted example.	2.88	<u>3.88</u>

The difference in higher and lower number refers to the inversion of the Likert questions in the questionnaire. The better number (higher user satisfaction) has been written in shaded cells. All the average answers included in the table have a significant difference between task one and task two.

In the post-experiment questionnaire the preference of the tool in task two was clear. Some of the users realised that the tool in task one has some power but they could not discover it because it inferred constraints using difficult examples as appear from the first question in the Table 6-15. Most of the user believed that the tool in task two is adapted to them and it thinks in the same way they are doing and they have done better (higher performance) in the second task than the first.

Table 6-15: Post-experiment questionnaire.

Question	Average answer (higher number = agree out of 5)
In task1, if I had used different examples, the tool would have been able to infer the correct constraint.	3.0
In the second task, the tool thinks like the way I think.	4.1
The tool was better able to define constraints in the second task than in the first task.	4.7
The tool learned how to define the constraint.	4.4
Task two was easier than task one.	4.7
It was easy to add a rule using the wizard.	3.1
The Rule Addition feature was useful.	4.4
I did better in task two because I added rules in task one.	4.8

Most of the participants agree that the tool learned how to define the constraint possibly because they have seen the tool generalising in task two and they believed to an extent that using the wizard is not so difficult compared to the benefit and usefulness of the adding rule feature.

6.4.4.5 Discussion

As can be seen in the above results, there was a higher level of correctness, less rules added and less time required accomplishing the tasks in task two than in task one. In all cases, there was a significant difference between both tasks in all the measured criteria. Therefore, the learning technique has been demonstrated to be feasible and potentially advantageous when using CSBE.

As has been introduced above, any constraint that required the use of “adding rule” feature was considered to be incorrect. However, in 4 cases (user 10, task one, constraint 1; user 10, task two, constraint 6; user 14, task two, constraint 5; user 15, task two, constraint 5), there was an incorrect selection for the constraints. This means that the user selected a incorrect constraint from the inferred list in the sense that it did not correspond to the constraint specified in the task list. If these 4 cases are ignored, it is possible to detect that the correctness figures (Figure 6-16 and Figure 6-17) is a mirror reflection of the adding rules figures (Figure 6-18 and Figure 6-19), the participant and constraint figures. These results reflect the importance of the learning technique both in cases where the constraint example is already in a rule in inference engine and in cases where it is not. The increase of correctness and reduction of the number of rules added in task two indicate that using the learning technique has been successful in adapting the system to the user.

In almost all the cases when the user required to use the “adding rule” feature in task two, this was because the user introduced a complicated example (that cannot be thought of as a prototype to express the constraint) at the teaching stage and then subsequently forgot how s/he taught the system to infer the constraint. This is inferred because the users who made mistakes (defined a constraint wrongly) in task two were doing these mistakes because they introduced different examples than those they used to teach the system in task one to specify similar constraints. It is believed that using very simple examples (simple in this context means the examples that may

come directly to mind when thinking of the constraint as prototype examples) to teach the system is the best way to remember the example later on. This feature has not been implemented to increase the complexity of the user's task by forcing remembering the example used during teaching stage. Instead, it is implemented to facilitate expressing examples in CSBE and to reduce the memory requirement. Otherwise, the user would be asked to remember how the system infers the constraints. This adapts the user to the system, the thing which this feature aims to avoid.

It was also observed during the experiment that the user hardly noticed that both constraint lists are similar. This can be explained in a couple of ways. First, there is a completely different constraint in each list, viz., the two label-related constraints discussed before in Section 6.4.3.5. Second, modifications were introduced to the constraints in the two lists to test the generalisation feature. Finally, there was a period of time, about 5 minutes, between finishing the first task and starting with the second which is used in filling in one of the questionnaires. If this period is added to the period after the user finished the constraint itself in the first task, this leads to an average gap of 17.6 minutes between specifying each constraint in the first task and the similar one in the second task. This is one of the reasons why the constraints were placed in one order in both tasks. In general, the fact that users could specify the constraints correctly in task two, even though they didn't recognise the similarity to constraint specifications in task one, provides evidence for the learning technique adapting the system to the user

The significant difference between task one and task two regarding the time required to accomplish the task can be explained as a direct effect of the learning technique. Since the system is adapted to the user and "thinks like them" in task two but not in task one, then it is to be expected that the user spends less time to accomplish the latter task. Also, the user does not need to spend time searching for an example to express the constraint in task two. This is because the system infers the constraint from the user's original example which has been taught to the system in the first task.

Responses to the questionnaires shows that the users preferred task two because they have the feeling that the system is adapted to them and they can specify

constraints using their preferred examples to express these constraints. They believe that the tool is adapted and thinks like them with an average of 4.13 out of 5. They also believe that in the second task the tool has learned how to define the constraints taught in the first task with an average of 4.38 and the learning technique was efficient with an average of 4.25. They all agree that task two was easier and the “adding rule” feature is useful because they agree that they did better (higher performance) in task two as a result of the “adding rule” feature. Surprisingly they believe that the learning (rule addition) process using the wizard is difficult with an average of only 3.63. One of the questions asks if they believed they got answers from the tool even for different constraints where “different constraints” here means possessing different properties. The average answer of this question is 4.06. This means that they knew that the two lists are not identical and they considered the constraints different because of the different properties. This answer also indicates that they liked the generalisation feature and understand its benefit.

It might be argued that the study depends on comparing the performance before and after teaching the system without taking into consideration the effort spent in teaching. If the effort that the user spent teaching the system is taken into consideration and added (as time) to the second condition (after the system has learned), the results will be different. This argument depends on the observation that the user is spending time and effort in teaching the system and this effort should be taken into consideration in the analysis of the results. However, in response, it should be pointed out:

- When the user is trying to define a constraint using CSBE and the system does not have the knowledge to infer the required constraint and specify it, the user must use the form-filling technique whether or not the system learns to infer the constraint. Since the user must use the form-filling technique in all cases including this effort in the comparison is unfair. This point was clarified earlier when the learning feature was discussed in Chapter 4. The discussion in the earlier chapter focused on the fact that learning is an activity that the system performs during the normal work of the user, viz., when using the form-filling technique to define a new constraint.

- When the user uses the form-filling technique to specify a constraint and the system learns, this is done once. If this is considered as a cost the user pays to teach the system, then this cost and effort will be reduced each time the user is specifying the same constraint or a similar constraint. This means that the effort should be divided and distributed over all the times that the user uses the added rule. This effort and its reduction over time and use of the added rule can be evaluated using a study that involves using the system for longer time and for different diagram types.

Of course, adding one rule or customising the inference engine for a specific diagram type does not mean that later on the user cannot add or perform more customisation to the inference engine for the same diagram type. Based on the above results and discussion, it is possible to reject the null hypothesis and accept the alternative hypothesis stating that “there is a significant difference between task one and task two regarding the measured criteria.”

6.4.5 Threats to Validity

One major threat to validity of this experiment is that the researcher performed the constraint specification using the DECS wizard when a constraint required to be added. This surely affected user perception of the difficulty of task one, making it appear not particularly difficult. This appears in the answers in the questionnaire regarding this part. However, if they were left to do this task by themselves, their opinion would most likely be more severe regarding the difficulty of task one.

The overall aim of this study was to investigate whether the rule addition technique can improve the performance of CSBE. The rule addition system, as it stands, offers a potential improvement for *part* of the process but must still fall back on a non-CSBE technique for the other part of the process (i.e., actual definition of the constraint). Requiring the participants to use the wizard would have had several adverse consequences for the experiment:

- The experimental design would have been complicated by the need to include wizard use and training.
- The time to perform the experiment would have been considerably increased, with consequent problems of participant recruitment and fatigue.

- Results related to features of the rule addition technique, apart from the wizard component, could have been influenced by participant performance with, and reaction to, the wizard itself.

For these reasons, it was decided to remove wizard use from the experimental design, with the researcher using the wizard when necessary during each trial. Evidence that this decision removed, or reduced, the effect of the wizard on the overall study results is the fact that there is no significant difference between the two tasks for the question “How hard did you have to work to accomplish your level of performance?” This insignificance indicates that the user was not aware of the additional effort of wizard use in the process.

Another threat could arise from a participant learning effect from the first to the second task. This may threaten the validity of both time and correctness data as the user may have learned from task one and consequently improved their performance in task two, since both tasks have sets of similar constraints. The experimental design attempted to reduce this effect as follows:

- Participants were not told during training that the same, or similar, constraints would be used in both tasks.
- Putting similar constraints in the same order in each task list leaves the maximum average time gap between a constraint in the first list and the one that is similar to it in the second list.
- Changing properties in related constraints hides their similarity.

6.4.6 Related work

Learning has been introduced in different literature as introduced before. However, the learning introduced here in this research is different from most of that reported in the literature on PBE (see Section 2.10). Learning in CSBE is a meta-level learning of how to infer specific constraints from customised or added examples. In other words, it consists of the system learning the way its user thinks and prefers to express constraints using examples. The only literature of which the author is aware that documents a similar idea is Myers et al. (2000) who claimed that it would be

desirable to be able to augment the rules of a PBE inference engine at run-time without the need to manipulate code manually.

Castelli, Oblinger, & Bergman, (2007) introduced learning in DocWizard, but it is a different concept than that introduced in DECS. Their learning concept is similar to that introduced in MetaMouse (Maulsby & Witten, 1993) in which the user is required to correct a wrong inference of the system. This learning technique depends on combining positive and negative examples to refine the behaviour of each other which requires extensive involvement of the user. However, DocWizard is not implemented to use positive and negative examples; instead, it depends on “incremental update” of the generated documentation based on the subsequent different actions of the users. Such systems are usually complicated because they are sensitive to every change in user behaviour. This means that every action of the user is counted and if the user makes a mistake the generated program will be difficult to recover; therefore, mistakes are not allowed in such systems. The generalisation feature implemented in the learning technique presented in this section, that infers the required constraint from similar examples, is entirely different from DocWizard and MetaMouse, has not been used before in any PBE system.

Myers, McDaniel, & Wolber (2000) and Castelli et al. (2007) encourage involving the user in the process of PBE. It has been discussed before that DECS involves the user in the process of inference through introducing the example and selecting the required interpretation and then selecting the required constraint from the inferred constraint list. DECS also involves the user in the process of learning via interaction with the “adding rule” feature. This is through offering the example and specifying the associated constraint using the wizard.

Chapter 7

Summary,

Conclusions and

Future Work

7.1 Introduction

This chapter presents the contributions achieved in this research and summarises the thesis argument. The chapter also offers conclusions in terms of the original research questions as well as the overall aims of the research and offers recommendations for potential applications of the results for meta-CASE tools and CSBE. Finally the chapter sets out several possible directions for this research topic in the future.

7.2 Research Contributions and Achievements

This research produced several contributions to software engineering knowledge. The following can be considered as the major contributions of this research:

Table 7-1: Contributions distributed over thesis chapters

Chapter	Contributions
Chapter 4-a	<ul style="list-style-type: none">• A novel specification technique, CSBE, for constraint specification in meta-CASE tools.
Chapter 6-a	<ul style="list-style-type: none">• Demonstration, via an empirical study, that CSBE is superior to a wizard-based form-filling technique.
Chapter 6-b	<ul style="list-style-type: none">• An empirical study of the relative comprehensibility of constraints expressed negatively vs. those expressed positively in a natural language.
Chapter 6-c	<ul style="list-style-type: none">• Demonstration, via an empirical study, that the use of a multi-polarity technique vs. a uni-polarity technique improves CSBE performance.
Chapter 6-d	<ul style="list-style-type: none">• Development of a novel rule augmentation technique for CSBE. <p>AND</p> <ul style="list-style-type: none">• Demonstration, via an empirical study, that adding a rule augmentation facility to a CSBE system improves performance.

The above contributions were supported by a number of additional achievements, including:

Table 7-2: Achievements distributed over thesis chapters.

Chapter	Achievements
Chapter 2	<ul style="list-style-type: none"> • A literature review of: <ul style="list-style-type: none"> ◦ meta-CASE tools, their use of constraints, the characterisation and classification of constraints and the methods by which they are defined, ◦ PBE with its different application contexts, use of example polarities and techniques for rule learning.
Chapter 3	<ul style="list-style-type: none"> • Enhancements to the DECS meta-CASE system which made it suitable to be used as a prototype for this research. • Design and development of an XML-based constraint language used in the studies in this research. This language has many features, such as flexibility, which qualified it to be adopted in this research. • Development and implementation in DECS of a constraint management component that handles constraints specified in the constraint language described above.
Chapters 4-7	<ul style="list-style-type: none"> • Implementation of CSBE in DECS with all of its distinctive features.

7.3 Thesis Summary

This research has the aim of simplifying and facilitating constraint specification within the meta-modelling process in a meta-CASE tool. A solution to the difficulty of constraint definition is proposed based on using a new technique called Constraint Specification by Example (CSBE), adapted from the well-known technique, Programming by Example (PBE).

This dissertation started with an introduction that presented the problem and set out the aims and objectives of the research. A solution was proposed and the approach to achieve the aim and objectives of the research was presented. Finally, a roadmap for the whole dissertation, annotated with contributions, was introduced.

The second chapter of the dissertation reviewed relevant literature on meta-CASE tools, their limitations, the challenges of constraint specification. It also reviewed Programming by Example, identifying its key features and variants.

Chapters 3 and 4 set the context for the research and presented CSBE, a novel constraint specification technique. Since a meta-CASE tool, Diagram Editor Constraints System (DECS), had already been developed in Glasgow University's School of Computing Science, it was adopted and enhanced (as described in Section 3.2.3) to be a suitable platform for conducting this research. To be able to define constraints, a flexible XML-based constraint language was developed and implemented.

Based on Programming by Example, CSBE was developed as a novel constraint specification technique that addressed the limitations of current meta-CASE tools. An additional form-based specification technique, using a wizard and tabbed forms, was designed and implemented in DECS as a representative of state of the art constraint specification in meta-CASE tools, to be used as a control in subsequent empirical studies of the performance of CSBE.

To achieve the research aim, it was necessary to test the main claim in the thesis statement: *"It is possible to simplify and facilitate the constraint specification process in a meta-CASE tool using the CSBE technique"*. Testing this statement was broken down into answering three key questions:

Does CSBE improve the performance of constraint specification in a meta-CASE tool compared to the form-filling technique?

Does example polarity influence the performance of CSBE?

Does implementing and using the learning technique influence the performance of CSBE technique?

Answering these three questions was achieved through four empirical studies summarised below.

Does CSBE improve the performance of constraint specification in a meta-CASE tool compared to the form-filling technique? Chapter 5 introduced the first

study (Study One) that was conducted to answer this question. In this study, the form-filling technique, which has been used in some meta-CASE tools before, was compared against CSBE. The experiment measured three criteria to represent the performance of the techniques: correctness, time required, and user satisfaction. Results showed that CSBE is superior to the form-filling technique with respect to all three criteria. Since the constraints can be expressed using positive or negative examples, some users commented that it would be useful to allow users to explicitly select the polarity of an example. This raised the second question and motivated studies three and four.

Does example polarity influence the performance of CSBE? To answer this question, chapter 6 introduced a study (Study Three) that has been conducted to evaluate the effect of example polarity on the performance of the CSBE technique. To achieve this general aim, two DECS implementations were prepared, the first supporting multi-polarity examples and the second allowing only uni-polarity examples. To provide the system with the ability to infer the required constraints and to provide a fair comparison between both implementations, the uni-polarity tool also included constraint specification by action to increase its expressiveness.

To investigate the potential confounding effect of the linguistic expression of constraints in Study Three, a preliminary study (Study Two) was conducted using an online questionnaire to test the preference of the users for the polarities in natural language constraint expression. Results of this study showed that there is a statistically significant preference for the positive expression of constraints; this led to appropriate expression counterbalancing in Study Three to deal with this effect.

The third study (Study Three) was conducted to compare between multi-polarity and uni-polarity constraint example tools described above. The study measured performance in terms of the number of constraints correctly defined using each tool, the time required to complete the task of constraint definition, and user satisfaction elicited through several questionnaires. Results showed that allowing the expression of constraints using multi-polarity examples outperformed uni-polarity examples with respect to all the measured criteria.

Does implementing and using the learning technique influence the performance of CSBE technique? In the above two studies, the constraints in CSBE are inferred based only on previously defined examples. This created a problem if the user does not think in the same way as the tool developer; namely, the user will not be able to express the required constraint. To solve this problem, there was a need to implement a technique for augmenting rules in the inference engine to customise and personalise the tool according to user preference. The third question stated above asks whether or not this feature adds value to CSBE. To answer the question, chapter 7 introduced a study (Study Four) conducted using a customisable version of DECS with an inference engine that cannot infer some constraints and can infer some others only with examples that are potentially difficult to imagine (according to the researcher). Participants were asked to define constraints using the tailorable version CSBE in two stages; inference rules added in the first stage could be used in the second stage. Measures included number of constraints defined correctly, time to accomplish the tasks and user satisfaction. Results showed that the rule addition feature positively affected the performance of CSBE in all three measures.

7.4 Conclusions and Recommendations

The results of Study One answered the first question,

Does CSBE improve the performance of constraint specification in a meta-CASE tool compared to the form-filling technique?

The study results demonstrate that the CSBE is more effective and efficient than the form-filling technique in DECS. In other words, CSBE facilitated the constraint specification. CSBE facilitated the constraint specification task because it reduced the error associated with the constraint specification task and increased the number of constraints specified correctly. CSBE also reduced the time required to accomplish the constraint specification task and bridged the gap between the specification and application formats. The last is achieved through visualisation of the constraint specification task and through the distinctive feature of specifying constraints of the target language using its own visual elements (vertices and edges) instead of using a representation, or expressing them, in other formats.

The results can be generalised to conclude that the CSBE technique constraint specification in the domain of meta-CASE tools in general, not just in DECS. DECS shares the key common features definitive of meta-CASE tools, including the basic meta-modelling process (basic because it does not depend on graphical meta-models), specification and generation of customised modelling languages, and more importantly, specification of constraints. In particular, almost all the meta-CASE tools depend, in different ways, on constraint specification within their meta-modelling process. Consequently, simplifying constraint specification using CSBE should be of potential benefit to the entire class of meta-CASE tools.

This generalisation is also supported via previous research that demonstrates that features used in CSBE add value to the specification process. For example, Goldman & Balzer (1999) claim that visualisation simplifies the process of expressing editor vocabulary and the GUI. As discussed in Sections 2.7.1 and 2.8.1, they call this “specification-by-example”. Although it is not clear exactly what they mean by ‘example’ in this phrase, it appears to refer to the visual representation of the meta-model they construct and the definition of the vocabulary and GUI of the target editor. Recalling again here that they import constraints in the form of programs called “analysers”. Therefore, the term ‘example’ in Goldman & Balzer’s notion of specification-by-example does not include constraint specification.

Another example is Draheim et al. (2010) who document that using the same visual objects that are to be specified to construct the specification increases the intuitiveness. The results can be generalised even to conclude that CSBE technique facilitates and simplifies the process of diagram editors or DSLs specification. This conclusion is based on the facts that CSBE facilitates constraints specification and the fact that constraints is part of the meta-modelling process. Since CSBE facilitates the constraints specification which is part of the meta-modelling process that generates a meta-model specifying a modelling language, then CSBE facilitates specifying the target modelling languages or diagram editors. However, this conclusion needs extra research to be validated. These conclusions recommend and support the application of CSBE technique for constraint specification purposes in meta-CASE tools. Some authors have introduced visual specification of meta-models, such as Goldman & Balzer (1999) who specify the vocabulary and GUI for

the required diagram editor. Others, such as Minas & Viehstaedt (1995), Rekers & Schurr (1995), and Lewicki & Fisher (1997), make use of visual languages. However, these languages were restricted to the description of the graph formalism such as hypernodes and hypergraphs; their effort was focused on developing parsers for the graphs to be able to specify the required editors. The point is that in all of these meta-CASE tools, constraints were specified using a textual format.

In addition, since CSBE has facilitated constraint specification in the domain of meta-CASE tools, it is possible that this technique would also facilitate constraint specification in other domains that depend on constraints. An example of such domains is a timetable management system used to arrange a timetable based on some constraints. These constraints could be specified using CSBE and might be particularly valuable since the users of a timetable system would probably not be expert in computing and they would need an intuitive method for constraint specification.

Thus, the first research question can be answered positively.

Results of Study Two and Study Three answered the question,

Does example polarity influence the performance of CSBE?

One can conclude from the results that constraints expressed positively in a natural language are perceived as easier to understand than constraints expressed negatively. Results also support the claim that providing the user with the ability to express constraints using either positive or negative examples increases the performance of CSBE compared to allowing negative examples only. In general, it appears that

- Offering a user more alternative ways of expressing a constraint improves performance.
- Some constraints are easier to express positively and others negatively.
- People prefer different expressions to express the same constraint.

A uni-polarity approach, although powerful enough to express all the required constraints, limits alternatives and provides only one way of specifying the constraints

which increases the difficulty of expressing the constraints that are easier to be specified using the opposite polarity examples. Almost all the previous research has introduced the positive and negative examples for specification but not as separate concepts; instead, they were introduced as one concept to refine the behaviour specified by each other. Based on the studies presented in this dissertation, it is reasonable to recommend offering the ability to express constraints using multi-polarities examples in meta-CASE CSBE as well as similar application domains. These conclusions and recommendation answers “Yes” to the second research question.

Study Four addressed the question,

Does implementing and using the learning technique influence the performance of CSBE technique?

One can conclude from the results that implementing a technique that allows the inference engine to be augmented with new rules and personalised examples improves the performance of constraint specification using CSBE. From the study the generalisation feature associated with the adding and customising rules transforms the process of rule augmentations and customisation into a learning technique. Accordingly, it is possible to conclude that using a learning technique that allows the system to build on experience of the user improves CSBE performance.

A further generalised conclusion is that a learning technique, whatever this technique is, which leads to augmenting the system by experience and customisation, positively affects the performance of constraint specification in CSBE. The last conclusion needs some extra research to validate it; however, since one learning technique was successfully implemented and tested in the research, it is believed that other techniques will also have similar successful results. This claim is based on the use of a learning technique in the DocWizard PBE system (Prabaker, Bergman, & Castelli, 2006). Based on the results and conclusions from this study, it can be recommended to implement a feature for adding rules and customising the inference engine of any constraint specification system that depends on CSBE or any other PBE technique. The recommendation can be extended to provide the ability for meta-meta level definition of rules and customisation in the form of a learning technique in any

constraint specification technique that depends on CSBE technique or any similar PBE technique. This answers the third question as “Yes”.

One might argue that a system that provides well-designed rules could be more robust and comprehensive than one that adds rules in an ad hoc manner. Clearly, a well-designed and “generative” visual language would appear to be superior to an arbitrary set of inference rules. However, with a fixed set of rules, even ones that are well-designed and comprehensive, the customisation of a tool to specific users and contexts is difficult, if not impossible.

The alternative followed in this research was to design a core fixed set of rules and assume they are sufficient as a basis to teach to users. Although learning and understanding how to express constraints using examples may be easier than learning a constraint programming language, it will end up involving learning and recalling specific examples, that were designed by somebody (the designer) who may conceptualise constraints in a way other than the user. Fixed rules are a good thing in languages, particularly formal ones, but the ability to add rules and customised examples has been designed to support the personalisation of the tool and the potentially idiosyncratic examples that express the constraint. In other words, creating the tool that understands how the user thinks and adapts to his/her thinking instead of adapting the user to the tool rules.

From the above, in general it is possible to conclude that CSBE proved its value in the constraint specification task within the domain of meta-CASE tools. It improves the performance, reduces the complexity and facilitates the constraint specification task which is part of any meta-modelling process for the purpose of CASE tool specification. Consequently, CSBE helped in addressing the research problem introduced at the start of this dissertation. CSBE’s distinctive features support the constraint specification process; they improve specification performance through increasing effectiveness and efficiency and the user’s experience of specification. Two of its features, providing the ability to use multi-polarity examples and the ability to learn and personalise the inference rules, have been evaluated and shown to be valuable.

Based on the above, it can be concluded that that *CSBE simplifies and facilitates the constraint specification process in meta-CASE tools.*

7.5 Future work

During the work reported in this dissertation, many ideas came up that are suitable for the future development of the research. These ideas fall into two categories. The first is the development of new features and the enhancement of the current implementation of DECS. The second is in the category of conducting additional empirical studies and research on the newly invented technique in this research, CSBE.

This section introduces these future work ideas. It presents the possible future work ideas one by one, starting from the work that can be conducted in the early future (within a year) as research or enhancement of the current version of DECS and based on the result of this research.

7.5.1 More Diagram types to be Involved in the Research

In this research there was a concentration on one diagram, the state transition diagram, for the conduct of experiments. Use case diagrams were also used in Study One and were used for training purposes in the other studies wherever training was required. The first future work proposed is to use more diagram types that require more user's experience in the diagram types and more complicated constraints than those used in this research (i.e., constraints that involve more vertex and edge properties such as the details of attributes and methods names in a Class vertex type in a Class diagram). Although the constraints used are typical and used in many other modelling diagram types and domain specific languages, it is possible that some diagram types, such as Data Flow Diagrams (DFD) and Entity Relationship (ER) diagrams, might require more complicated constraints.

To include more complicated diagram types (i.e., those that have several properties and concepts) such as class and package diagrams, DECS itself must be updated to be able to model these diagram types with all of their required features, such as the ability to specify constraints over attributes of the class diagram and over

inclusion in package diagrams. A DECS version which is able to model sub-diagram inclusion was implemented as a Masters project at Glasgow University towards the end of this research (Calisti, 2010). Another Masters project to enhance DECS' class diagram modelling is currently underway. These versions could be used as a starting point for this proposed work. In addition to improving the modelling capabilities of DECS, such research would also open the way to enhancement of the constraint language introduced in this research.

7.5.2 Enhancing the Constraint Language

The constraint language used in this research has many distinctive features as discussed previously. However, it is still immature and needs to be developed. One of the implementation advantages of the language is its extensibility. To extend the language the following steps are generally required:

- Extend the XML tag set as required. No extra implementation is required for the parser as it is designed to parse the files with its concepts such as the URI references.
- Extend the factory method required to build the Java object representing the constraint by adding any new attributes in the XML.
- Add the required Java classes to build the constraint and check it. This implements the required behaviour of the added features.
- Redesign the wizard to include the required GUI to capture the new features.
- Extend the inference engine class with the required rules to infer new constraints from examples. This also requires extending the factory mapping method that maps the rules into Java classes that generate the inferences. This last part can be ignored if all the new examples and constraints will be taught to the system. However, the features that should be extracted from the model must be implemented as rules.

It is believed that the constraint language should be developed gradually to be able to capture the concepts that different modelling language specification may require.

7.5.3 Comparing CSBE with other constraint specification techniques

In this research CSBE has been compared with the form-filling technique only. However, there are other techniques for constraint specification such as spreadsheets and visual languages. This requires further development and implementation of such techniques in DECS to be able to conduct research comparing them with CSBE.

7.5.4 Ranking the Inferences

Assume that a user wishes to specify the constraint “*It is not allowed to connect two Start State vertices using Transition edge*”. Assume the user introduced a negative example of 2 start states connected by a transition edge and the system infers the constraint. The system will not only infer the required constraint from the example; instead, it infers several other constraints and presents them in a list. Assume it inferred 5 constraints and the required one is the first in the list. Assume now that the user needs to specify another constraint, for example, “*Start States must not have incoming edges*”. The user may use the previously introduced example for the first constraint to specify the second constraint. Assume the system infers the required constraint and it is ranked 4th in the list. This means that the user needs to read the first constraint, which is the one already specified, reads the second, the third and then finds the solution when reading the 4th. It is possible to imagine the situation of defining 3 or 4 constraints using the same example and the user reads the same constraints in the list over and over.

This problem can be solved by providing checkboxes to select the required constraints from the inferred list instead of using radio buttons. This could be a good solution but it is believed that the user when specifying the constraints is focusing on the constraint that is to be specified instead of looking to the whole picture of the language to be specified. However, this is a claim that needs more research to prove. Another solution is implementing a learning system that provides different rankings for the inferences in the list based on the previous experience and the current user task. This system should provide two different behaviours. The first is to rank the selected constraints. This means that the system puts the previously selected constraints at the bottom of the list so the user is not bothered with reading them

again. This behaviour should be followed throughout the same session of defining a modelling language. However, when the user provides a similar but not exactly the same example (the vertex types is different, for example), then the system should behave in the opposite way. This means that the system should rank the previously selected constraints more highly because it is more likely that the user is defining the *same* constraint for some other component types. The system also should follow the same “ranking up” behaviour when the user finishes defining the language s/he is working on and starts another session for specifying another modelling language (this is possibly sometime later). In this case the system should assume that the user is going to specify similar constraints that were specified in the previous modelling language. In other words, the system assumes that these constraints are the most used once so it ranks them at the top of the list.

Ranking constraints may have little effect when the constraints generated from an example are few and are described using short sentences. However, it is believed that it has a higher impact and importance if the number of constraints generated from an example is high and the sentences describing the constraints are long. Again, this is just a claim and needs extra research to confirm. This ranking feature has not been noticed or documented in any PBE system, not even as a future work, in the literature reviewed in this research. A ranking approach that provides multiple ways of ranking constraints has been introduced in the PhD thesis of T. McBryan (2011). This idea could be adapted to CSBE and its feasibility and desirability in the domain of meta-CASE tools can be studied.

7.5.5 Recommendation System Depending on Previous Specifications

This suggestion is related to the work introduced in the previous section. The inference system could collect information about the constraints that are selected during a modelling language specification session. This information could be considered as experience for the system and this experience could be accumulated over time and for every modelling language specified. The system supporting this experience collection should be able to find relationships between the selected constraints for different languages. These relationships allow the system to suggest possible constraints based on the previous experience of specifying the modelling languages.

This work depends on the claim that there are many shared constraints between different modelling languages. This means that if 10 constraints are defined to specify the modelling language **A** and one of these constraints is selected while defining the modelling language **B**, then there is a possibility that some of the other 9 constraints of language **A** are relevant for language **B**. This feature could improve the performance of defining the constraints for a language; however, it requires a feature-based pattern matching system to recognise similarities among the selected constraints since component types are different between modelling languages. Such a technique could be the basis of a new constraint specification technique called ‘specification by suggestion’ or ‘specification by recommendation’. A recommendation system could be used or adapted to achieve this work such as that introduced in (McBryan, 2011).

7.5.6 Enhance Rule Addition by Selecting the Required Collected Features

This future work is one of the enhancements that can be added as a feature to DECS’ rule addition, or learning, technique. When the user introduces an example and the required constraint is not inferred, the user chooses to add an inference rule. At this point the system collects features from the introduced example in the form of rule triggers. These rule triggers are saved with the added rule so when this rule is fired again in the future because of an introduced example, the added rule will also be fired as it is triggered by a set of features in the example that are satisfied. With this scenario, the generalisation and the control of the added rules are fixed by the system configuration.

Such a facility might operate by showing the user the potential rule triggers (example features) that will be saved with the added rule. This set of triggers will appear in a list, each trigger with an associated checkbox. The user then can select the required features to be taken into consideration and required to trigger the newly added rule. This gives the user the ability to restrict or relax the generalisation of the added rule.

When the user restricts the added rule to the maximum this means that the rule will only be fired when exactly the same example is introduced again. In this case all the fired rules and collected features from the example including the component types should be taken into consideration. DECS does not have the ability to specify the

specific vertex or edge types in the rules in the current implementation; however, for maximum restriction this needs to be implemented. It is believed that maximum restriction (i.e., no generalisation) doesn't need to be added as a rule since the general rule can infer the specific example but not vice versa; in general, however, this is one of the possible degrees for generalisation. The other extreme of generalisation is to consider the lowest possible number of features for generalisation. The ability to control the learning degree through controlling the generalisation was introduced in the DocWizard system (Castelli, Oblinger, & Bergman, 2007).

7.5.7 Visual Language

Since DECS depends on an XML-based constraint language that has attributes with values, it is possible to visualise this language in form of a basic visual representation (e.g., squares, oval shapes and circles). The XML file that defines a constraint can be represented in form of circles each of which is an attribute in the constraint. The value for each attribute can be given in form of a label added to the circle. This representation will be similar to an Entity Relationship diagram. Entities in the diagram represent the main type of the constraint, graph, vertex or edge. These entities can be connected to other entities to represent referenced constraints. All the required features for each entity such as the type of the constraint (hard or soft), the upper bound number or any other feature can be attached also to the main entity. Although this technique requires the user to understand and learn the language and how to use the model to specify a constraint, such a constraint definition technique could be easier to use than DECS' wizard. A visual language has been used before for definition event handling (Li, Hosking, & Grundy, 2009). However, their language is more general and requires the user to learn the meaning of each component to define an event handler.

7.5.8 Constraint Conflict Checker

One of the advantages of using a formal constraint language, such as OCL, is the ability to check for constraint conflict or redundancy. Since DECS depends on an informal XML-based constraint language, it is not possible to use formal techniques to check for constraint conflict and redundancy; it is not impossible, however. As future work it is proposed to add a conflict checker to DECS by translating the

generated constraints into OCL, so that OCL-based constraint analysis tools can be used.

Bibliography

1. Abraham, R., & Erwing, M. (2006). Inferring Templates from Spreadsheets. *International Conference on Software Engineering*, (pp. 182-191). Shanghai, China.
2. Ackermann, J. (2005). Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. *MODELS Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends* (pp. 15-29). Montego Bay, Jamaica: ACM/IEEE.
3. Alderson, A. (1991). Meta-case technology. *Proceedings of the European symposium on Software development environments and CASE technology* (pp. 81-91). Springer-Verlag New York, Inc.
4. Ali, N., Hosking, J., Huh, J., & Grundy, J. (2009). Critic Authoring Templates for Specifying Domain-Specific Visual Language Tool Critics. *Australian Software Engineering Conference* (pp. 81-90). Gold Coast: IEEE.
5. Alpert, S. (1993). Graceful Interaction with Graphical Constraints. *IEEE Computer Graphics & Applications* , 82-91.
6. Amant, R., Lieberman, H., Potter, R., & Zettlemoyer, L. (2000). Visual Generalisation in Programming By Example. *Communications of the ACM* , 43 (3), 107 - 114.
7. Argall, B., Chernova, S., Veloso, M., & Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and Autonomous Systems* , 57 (5), 469-483.
8. Atkinson, C., Gutheil, M., & Kennel, B. (2009). A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Transactions on Software Engineering* , 742-755.
9. Barr, T. (2000). Experiences with the UML/OCL-Approach in Practice and Strategies to Overcome Deficiencies. *Tagungsband Net.Object Days 2000*, (pp. 192--201). Erfurt, Germany.

10. Bergman, L., Castelli, V., Lau, T., & Oblinger, D. (2005). DocWizards: A System for Authoring Follow-me Documentation Wizards. *Symposium on User Interface Software and Technology* (pp. 191-200). Seattle, Washington, USA: ACM.
11. Bergmann, G., Horvath, A., Rath, I., Vrró, D., Balogh, A., Balogh, Z., et al. (2010). *Incremental Model Queries over EMF Models*. Retrieved 10 9, 2010, from <http://home.mit.bme.hu/~rath/pub/conf/beta/models10-submitted.pdf>
12. Bimbo, A., & Vicario, E. (1995). Specification by Example of Virtual Agents Behaviour. *Visualization and Computer Graphics, IEEE Transactions on* , 350-360.
13. Bock, C. (2007). Model-Driven HMI Development: Can Meta-CASE Tools do the Job? *HICSS 2007. 40th Annual Hawaii International Conference on System Sciences, 2007.* , (pp. 287b-287b).
14. Bogie, N., Gray, P., Hamilton, P., McCallum, G., McGroarty, D., & Welland, R. (2000). *Design Editor Constraint System DECS 2000 : System Status Report*. Glasgow, UK: University of Glasgow.
15. Borning, A. (1986). Defining Constraints Graphically. *Conference on Human Factors in Computing Systems* (pp. 137-143). Boston, USA: ACM, USA.
16. Briand, L., Labiche, Y., Di Penta, M., & Yan-Bondoc, H. (2005). An Experimental Investigation of Formality in UML-Based Development. *IEEE Transactions on Software Engineering* , 833-849.
17. Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J., & Yang, S. (2001). ClassSheets Automatic Generation of Spreadsheet Applications from Object Oriented Specifications. *Jor* , 155-206.
18. Calinon, S., & Billard, A. (2008). A Probabilistic Programming by Demonstration Framework Handling Constraints in Joint Space and Task Space. *Proc. IEEE/RSJ Intl Conf. on Intelligent Robots and Systems (IROS)* (pp. 367 - 372). Nice, France: IEEE / RSJ.

19. Calisti, F. (2010). *Inclusion-based approach for the Meta-CASE tool DECS, MSci Project*. Glasgow: Glasgow University.
20. Castelli, V., Oblinger, D., & Bergman, L. (2007). Augmentation-Based Learning combining observations and user edits for Programming-by-Demonstration. *Knowledge-Based Systems* , 20 (6), 575-591.
21. Chen, J.-H., & Weld, D. (2008). Recovering from Errors during Programming by Demonstration. *IUI '08 Proceedings of the 13th international conference on Intelligent user interfaces* (pp. 159-168). Gran Canaria, Spain: ACM.
22. Clark, T., Evans, A., & Kent, S. (2003). Aspect-oriented Metamodelling. *The Computer Journal* , 566-577.
23. Costal, D., Gomez, C., Queralt, A., Raventos, R., & Teniente, E. (2006). Facilitating the Definition of General Constraints in UML. *MoDELS 2006, LNCS 4199* (pp. 260-274). Heidelberg: ACM/IEEE.
24. Cypher, A. (1993). *Watch What I Do: Programming by Demonstration*. Cambridge, Mass.: MIT Press.
25. De Lara, J., & Vangheluwe, H. (2002). Using AToM3 as a Meta-CASE Tool. *4th International Conference on Enterprise Information Systems (ICEIS)*, (pp. 642-649). Ciudad Real, Spain.
26. Dey, A., Hamid, R., Beckmann, C., Li, I., & Hsu, D. (2004). a CAPpella: programming by demonstration of context-aware applications. *Conference on Human Factors in Computing Systems, Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 33 - 40). Vienna, Austria: ACM, New York, USA.
27. Dictionary. (2011). *Oxford Dictionaries*. Retrieved 6 9, 2011, from <http://oxforddictionaries.com/definition/synergy>
28. Draheim, D., Himsl, M., Jabornig, D., Kung, J., Leithner, W., Regner, P., et al. (2010). Concept and pragmatics of an intuitive visualization-oriented metamodeling tool. *Journal of Visual Languages and Computing* , 157-170.

29. Ebert, J., Süttenbach, R., & Uhe, I. (1997). Meta-CASE in practice: A CASE for KOGGE. In A. Olivé, & J. Pastor, *Advanced Information Systems Engineering* (pp. 203 - 216). Heidelberg: Springer Berlin / Heidelberg.
30. Engels, G., & Erwig, M. (2005). ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications. *20th IEEE/ACM Int. Conf. on Automated Software Engineering* (pp. 124-133). IEEE / ACM.
31. Engstrom, E., & Krueger, J. (2000). Building and Rapidly Evolving Domain-Specific Tools with DOME. *Symposium on Computer-Aided Control System Design* (pp. 83-88). Anchorage, Alaska, USA: IEEE.
32. Ferguson, R., Hunter, A., & Hardy, C. (2000). MetaBuilder: The Diagrammer's Diagrammer. *Lecture Notes In Computer Science; Vol. 1889* (pp. 407 - 421). London, UK: Springer-Verlag.
33. Findeisen, P. (1994). *The Metaview System*. University of Alberta. Alberta: University of Alberta.
34. Fish, A., Hamie, A., & Howse, J. (2010). Visual Specification Patterns. *Electronic Communications of the EASST*, 31, 1-14.
35. Fisher, G., Busse, D., & Wolber, D. (1992). Adding rule-based reasoning to a demonstrational interface builder. *Proceedings of the 5th annual ACM symposium on User interface software and technology* (pp. 89-97). Monterey, California, United States: ACM .
36. Frank, M. (1995). Grizzly Bear: A Demonstrational Learning Tool for a User Interface Specification Language. *Proceedings of the ACM Symposium on User Interface Software and Technology* (pp. 75-76). ACM Press.
37. GME. (2005). *A Generic Modeling Environment GME 5 User's Manual*. Retrieved 5 12, 2011, from <http://w3.isis.vanderbilt.edu/Projects/gme/GMEUMan.pdf>.
38. Gliffy. (2011, 7 16). *Gliffy*. Retrieved 7 16, 2011, from Gliffy: <http://www.gliffy.com/>.

39. Goldman, N., & Balzer, R. (1999). The ISI Visual Design Editor Generator. *Visual Languages*, (pp. 20 - 27). Tokyo, Japan.
40. Gong, M., Scott, L., & Offen, R. (1997). MetaBuilder: a generic CASE tool builder. *Software Engineering Conference, 1997. Asia Pacific and International Computer Science Conference 1997. APSEC '97 and ICSC '97. Proceedings*, (pp. 435-444).
41. Gray, J., Bapty, T., Neema, S., & Tuck, J. (2001). Handling Crosscutting Constraints in Domain-Specific Modelling. *Communication of the ACM* , 44 (10), 87-93.
42. Gray, P., & Welland, R. (1999). Increasing the flexibility of modelling tools via constraint-based specification. *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. Mississauga, Ontario, Canada: IBM Press.
43. Groher, I., Reder, A., & Egyed, A. (2010). Incremental Consistency Checking of Dynamic Constraints. *Lecture Notes in Computer Science* , 203-217.
44. Grundy, J., Hosking, J., Huh, J., & Li, K. (2008). Marama: an Eclipse Meta-toolset for Generating Multi-view Environments. *ICSE, 08*, (pp. 819-822). Leipzig, Germany.
45. Grundy, J., Mugridge, W., & Hosking, J. (1998a). Static and Dynamic Visualisation of Software Architectures for Component-based Systems. *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering* (pp. 426-433). San Francisco: IEEE CS Press.
46. Grundy, J., Mugridge, W., & Hosking, J. (1998b). Visual specification of multiple view visual environments. *Proceedings of IEEE VL'98* (pp. 236-243). Halifax, Nova Scotia, Canada: IEEE CS Press.
47. Guo, Y., Sierszecki, K., & Angelov, C. (2009). Model-Driven Development of Domain-Specific Applications: Tool Support. *7th Nordic Workshop on Model Driven Software Engineering*, (pp. 225-239). Tampere, Finland.

48. Hamilton, P. (2000). *Increasing the Configurability of the Design Editor Constraint System*. Glasgow, UK: University of Glasgow.
49. Heffernan, N. (2003, May 1). *Proposal for the Research Advancement Program*. Retrieved October 14, 2009, from http://nth.wpi.edu/pubs_and_grants/Research_Development_Council_Proposal.htm
50. Henkel, M., & Stirna, J. (2010). Pondering on the Key Functionality of Model Driven Development Tools: The Case of Mendix. *Perspectives in Business Informatics Research, Lecture Notes in Business Information Processing* , 64, 146-160.
51. Hudson, S., & Hsi, C.-N. (1993). A Synergistic Approach to Specifying Simple Number Independent Layouts by Example. *Conference on Human Factors in Computing Systems* (pp. 285-292). New York, USA: ACM.
52. IBM. (2009). *IBM Rational Rose Enterprise*. Retrieved November 25, 2009, from IBM Rational Rose Enterprise: <http://www-01.ibm.com/software/awdtools/developer/rose/enterprise/index.html>
53. Iivari, J. (1996). Why are CASE tools not used? *Communications of the ACM* , 39 (10), 94-103.
54. Inglis, N. (2005). *Integration of DECS into the Eclipse IDE*. Glasgow: University of Glasgow.
55. Isazadeh, H., & Lamb, D. A. (1997). *CASE Environments and MetaCASE Tools*. Kingston, Ontario, Canada: Department of Computing and Information Science, Queen's University.
56. Jankowski, D. (1997). Computer-Aided Systems Engineering Methodology Support and Its Effect on the Output of Structured Analysis. *Empirical Softw. Eng.* , 2 (1), 11-38.
57. Jia, J. (2007). *DECS Project1, Evaluation and Enhancement*. Glasgow: University of Glasgow.

58. Jorgensen, M., & Sjoberg, D. (2004). Generalization and Theory-Building in Software Engineering Research. *Empirical Assessment in Software Engineering (EASE2004)*, (pp. 29-36).
59. Karsai, G., Nordstrom, G., Ledeczi, A., & Sztipanovits, J. (2000). Specifying Graphical Modeling Systems Using Constraint-based Metamodels. *Proceedings of the 2000 IEEE International Symposium on Computer-Aided Control System Design* (pp. 89-94). Anchorage, Alaska, USA: IEEE.
60. Kelly, S. (2009). *Domain-Specific Modeling: A Toolmaker Perspective*. Retrieved March 4, 2009, from MetaEdit+ Workbench questions: <http://www.metacase.com/faq/showfaq.asp?cate=MWB>
61. Kelly, S., & Tolvanen, J.-P. (2008). *Domain-Specific Modelling: Enabling Full Code Generation*. New Jersey: John Wiley & Sons.
62. Kelly, S., & Tolvanen, J.-P. (2000). Visual domain-specific modeling: Benefits and experiences of using metaCASE tools. *International Workshop on Model Engineering, ECOOP* (pp. 1-9). Sophia Antipolis and Canes, France: (ed. J. Bezivin, J. Ernst).
63. Kelter, U., Monecke, M., & Schild, M. (2009). Do We Need 'Agile' Software Development Tools? *Objects, Components, Architectures, Services, and Applications for a Networked World, Lecture Notes in Computer Science* , 2591, 412-430.
64. Kirchner, L., & Jung, J. (2007). A Framework for the Evaluation of Meta Modelling Tools. *The Electronic Journal Information Systems Evaluation* , 10 (1), 65-72.
65. Kleppe, A. (2009). *Software Language Engineering*. Boston: Pearson Education, Inc.
66. Koedinger, K., Aleven, V., & Heffeman, N. (2003). *Tools Towards Reducing the Costs of Designing, Building, and Testing Cognitive Models*. Retrieved 2 14, 2011, from http://nth.wpi.edu/pubs_and_grants/03-BRIMS-063.doc

67. Kristiansen, K. (2001). *A CASE Tool Builder*. Glasgow, UK: University of Glasgow.
68. Kurlander, D., & Feiner, S. (1993). Inferring Constraints From Multiple Snapshots. *ACM Transactions on Graphics* , 12 (4), 277-304.
69. Lau, T., Wolfman, S., Domingos, P., & Weld, D. (2001). Learning Repetitive Text-editing Procedures with SMARTedit. In Lieberman, *Your wish is my command: giving users the power to instruct their software*. Morgan Kaufmann.
70. Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., et al. (2001). Composing Domain-Specific Design Environments. *Computer* , 34 (11), 44-51.
71. Ledeczi, A., Maroti, M., & Volgyesi, P. (2001). *The Generic Modeling Environment*. Nashville, USA: Institute for Software Integrated Systems, Vanderbilt University.
72. Ledeczi, A., Maroti, M., & Volgyesi, P. (2004). *The Generic Modling Environment*. Retrieved 7 30, 2011, from GMEReport: <http://w3.isis.vanderbilt.edu/projects/gme/GMEReport.pdf>
73. Lewicki, D., & Fisher, G. (1996). VisiTile – A Visual Language Development Toolkit. *IEEE Symposium on Visual Languages* (pp. 114-121). Boulder, Colorado: IEEE Computer Society.
74. Li, K., Hosking, J., & Grundy, J. (2009). A Generalised Event Handling Framework. Auckland, New Zealand.
75. Liu, N., Hosking, J., & Grundy, J. (2007a). A Visual Language and Environment for Specifying User Interface Event Handling in Design Tools. *Conferences in Research and Practice in Information Technology Series; Vol. 241* (pp. 87 - 94). Ballarat, Victoria, Australia : Australian Computer Society, Inc., Australia.
76. Liu, N., Hosking, J., & Grundy, J. (2007b). MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism.

VLHCC (pp. IEEE Symposium on Visual Languages and Human-Centric Computing). Idaho, USA: IEEE.

77. Marttiin, P., Rossi, M., Tahvanainen, V.-P., & Lyytinen, K. (1993). A comparative review of CASE shells: A preliminary framework and research outcomes. *information & Management* , 11-31.
78. Maulsby, D., & Witten, I. (1993). Metamouse: An Instructive Agent for Programming by Demonstration. In A. Cypher, *Watch What I Do: Programming by Demonstration*. Cambridge: MIT Press.
79. McBryan, T. (2011). *A Generic Approach to the Evolution of Interaction in Ubiquitous Systems*. Glasgow, UK: University of Glasgow.
80. McCallum, G. (2000). *Adding Activity Monitoring and Visual Programming By Example to a Software Modelling Tool*. Glasgow: Department of Computing Science, Glasgow University.
81. McClelland, C. (2002). *Adding a replay facility to the DECS meta-CASE tool*. Glasgow, UK: University of Glasgow.
82. McDaniel, R., & Myers, B. (1999). Getting More Out Of Programming-By-Demonstration. *Conference on Human Factors in Computing Systems, Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit* (pp. 442 - 449). Pennsylvania: ACM.
83. MetaCase. (2009). *MetaEdit+*. Retrieved 7 5, 2009, from <http://www.metacase.com/>
84. Microsoft. (2010). *White Papers*. Retrieved August 19, 2011, from Microsoft Visio 2010: http://visio.microsoft.com/en-us/Get_Started/WhitePapers/Pages/default.aspx?Filter1=White+Papers
85. Minas, M. (2002). Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* , 44 (2), 157-180.

86. Minas, M., & Viehstaedt, G. (1995). DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams. *IEEE Symposium on Visual Languages* (pp. 203-210). Darmstadt, Germany: IEEE Computer Society.
87. Mugridge, W., Hosking, J., & Grundy, J. (1998). Vixels, CreateThroughs, DragThroughs and Attachment Regions in BuildingByWire. *OZCHI '98 Proceedings of the Australasian Conference on Computer Human Interaction* (pp. 320-327). Adelaide, Australia: IEEE CS press.
88. Myers, B. (1993). Peridot: Creating User Interfaces by Demonstration. In A. Cypher, *Watch What I Do: Programming by Demonstration*. Cambridge, Mass.: MIT Press.
89. Myers, B. (1986). Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. *CHI Proceedings* , 59-66.
90. Myers, B., McDaniel, R., & Wolber, D. (2000). Programming by example: Intelligence in Demonstrational Interfaces. *Communications of the ACM* , 43 (3), 82-89.
91. Nichols, J., & Lau, T. (2008). Mobilization by Demonstration: Using Traces to Re-author Existing Web Stites. *Intelligent User Interfaces*, (pp. 149-158). Gran Canaria, Spain.
92. Nikitas, C. (2005). *Graph-Type Specification and Design*. Glasgow: University of Glasgow, Department of Computing Science.
93. Nordstrom, G., Sztipanovist, J., Karsai, G., & Ledeczi, A. (1999). Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environment. *Engineering of Computer-Based Systems, 1999. Proceedings. ECBS '99. IEEE Conference and Workshop on* , (pp. 68-74). Nashville, TN , USA .
94. Offen, R. (2000). *CASE Tools and Constraints*. North Ryde: Macquarie University Joint Research Center for Advanced Systems Engineering.
95. Pohjonen, R. (2005). Metamodeling Made Easy - MetaEdit+ (Tool Demonstration). *Springer* , 442-446.

96. Prabaker, M., Bergman, L., & Castelli, V. (2006). An evaluation of using programming by demonstration and guided walkthrough techniques for authoring and utilizing documentation. *Proceedings of the SIGCHI conference on Human Factors in computing systems* (pp. 241-250). Montreal, QC Canada: ACM .
97. Qattous, H. (2009). Constraint Specification by Example in a Meta-CASE Tool. *Foundations of Software Engineering, Proceedings of the doctoral symposium for ESEC/FSE on Doctoral Symposium* (pp. 13-16). Amsterdam: ACM.
98. Qattous, H., Gray, P., & Welland, R. (2010). An Empirical Study of Specification by Example in a Software Engineering Tool. *Empirical Software Engineering Conference*. Bolzano, Italy.
99. Rekers, J., & Schurr, A. (1995). A Graph Grammar Approach to Graphical Parsing. *IEEE Symposium on Visual Languages* (pp. 195-202). Darmstadt, Germany: IEEE Computer Society.
100. Robbins, J., Hilbert, D., & Redmiles, D. (1997). Argo A Design Environment for Evolving Software Architectures. *Software Engineering, 1997., Proceedings of the 1997 (19th) International Conference*.
101. Robert, F., & Bernhard, R. (2005). Domain Specific Modeling. *Software System Model* , 4, 1-3.
102. Santos, A., Koskimies, K., & Lopes, A. (2010). Automating the construction of domain-specific modeling languages for object-oriented frameworks. *The Journal of Systems and Software* , 1078-1093.
103. Sazonov, E. (2004). *Open source fuzzy inference engine for Java*. Retrieved March 27, 2009, from Open source fuzzy inference engine for Java: <http://people.clarkson.edu/~esazonov/FuzzyEngine.htm>
104. Scott, L. (1997). Hypernode model support for software design methodology modelling. *Software Technology and Engineering Practice, 1997. Proceedings., Eighth IEEE International Workshop on [incorporating Computer Aided Software Engineering]*, (pp. 22-31).

105. Scott, L., Horvath, L., & Day, D. (2000). Characterising CASE Constraints. *Communications of the ACM* , 43 (11), 232-238.
106. Singh, G., Kok, C., & Ngan, T. (1990). Druid: A System for Demonstrational Rapid User Interface Development. *UIST '90 Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology* (pp. 167-177). SNOWBIRD, UTAH: ACM.
107. Sjoberg, D., Hannay, J., Hansen, O., Kampenes, V., Karahasanovic, A., Liborg, N.-K., et al. (2005). A Survey of Controlled Experiments in Software Engineering. *IEEE Transactions on Software Engineering* , 31 (9), 733-753.
108. Smith, D., Cypher, A., & Spohrer, J. (1994). KidSim: programming agents without a programming language. *Communications of the ACM* , 37, 54-67.
109. Sommerville, I. (2007). *Software Engineering*. Essex: Pearson Education.
110. Sommerville, I., Welland, R., & Beer, S. (1987). Describing software design methodologies. *Comput. J.* , 30 (2), 128-133.
111. Stevanovic, D. (2007, June 20). *Zigbee-Standard-Talk*. Retrieved April 18, 2011, from Youk University in Canada: <http://www.cse.yorku.ca/~dusan/Zigbee-Standard-Talk.pdf>
112. SurveyMonkey. (2011). Retrieved 3 2, 2011, from SurveyMonkey: <http://www.surveymonkey.com/>
113. Systems, I. f. (2011). *GME: Generic Modling Environment*. Retrieved 7 30, 2011, from GME: Generic Modling Environment: <http://www.isis.vanderbilt.edu/Projects/gme/>
114. Toomim, M., Drucker, S., Dontcheva, M., Rahimi, A., Thomson, B., & Landay, J. (2009). Attaching UI Enhancements to Websites with End Users. *ACM Conference on Human Factors in Computing Systems* (pp. 1859-1868). Boston: ACM.
115. Vessey, I., Jarvenpaa, S., & Tractinsky, N. (1992). Evaluation of vendor products: CASE tools as methodology companions. *Commun. ACM* , 90-105.

116. Wahler, M., Koehler, J., & Brucker, A. (2006). Model-Driven Constraint Engineering. *Proceedings of the Sixth OCL Workshop OCL for (Meta-)Models in Multiple Application Domains* (pp. 1-20). Electronic Communications of the EASST.
117. White, J., & Schmidt, D. (2005). Simplifying the Development of Product-line Customization Tools via Model Driven Development. *Workshop on MDD for Software Product-lines: Fact or Fiction? at the 8th International Conference on Model Driven Engineering Languages and Systems*. Jamaica.
118. Zhu, N., Grundy, J., & Hosking, J. (2004). Pounamu: A Meta-Tool For Multi-View Visual Language Environment Construction. *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing* (pp. 254-256). IEEE Computer Society.
119. Zschaler, S., Kolovos, D., Drivalos, N., Paige, R., & Rashid, A. (2010). Domain-Specific Metamodelling Languages for Software Language Engineering. *Springer-Verlage Berlin Heidelberg* , 334-353.

Appendix A

Constraint XML

Files

A.1 Vertex Constraint XML File Template

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<constraint name = "null" scope = "vertex" type =
"null" description = "null">
  <vertex>
    <type>
      <item>null</item>
    </type>
    <baseShape permission = "ignore">
      <item>null</item>
    </baseShape>
    <label>
      <text permission = "ignore">
        <item>null</item>
      </text>
      <textRE permission = "ignore">
        <RE>null</RE>
      </textRE>
    </label>
    <bgColour>
      <colour permission = "ignore">
        <item>null</item>
      </colour>
    </bgColour>
    <sourceConnections>
      <identicalLabels permission = "ignore">
        <value>null</value>
      </identicalLabels>
      <lowerBoundNumber permission = "ignore" value
= "null">
        <edge>
          <OR>
            <uri>null</uri>
          </OR>
          <AND>
            <uri>null</uri>
          </AND>
          <combination>
            <uri>null</uri>
          </combination>
        </edge>
      </lowerBoundNumber>
```

```

        <upperBoundNumber permission = "ignore" value =
"null">
            <edge>
                <OR>
                    <uri>null</uri>
                </OR>
                <AND>
                    <uri>null</uri>
                </AND>
                <combination>
                    <uri>null</uri>
                </combination>
            </edge>
        </upperBoundNumber>
    </sourceConnections>
    <targetConnections>
        <identicalLabels permission = "ignore">
            <value>null</value>
        </identicalLabels>
        <lowerBoundNumber permission = "ignore" value =
"null">
            <edge>
                <OR>
                    <uri>null</uri>
                </OR>
                <AND>
                    <uri>null</uri>
                </AND>
                <combination>
                    <uri>null</uri>
                </combination>
            </edge>
        </lowerBoundNumber>
        <upperBoundNumber permission = "ignore" value =
"null">
            <edge>
                <OR>
                    <uri>null</uri>
                </OR>
                <AND>
                    <uri>null</uri>
                </AND>
                <combination>
                    <uri>null</uri>
                </combination>
            </edge>
        </upperBoundNumber>
    </targetConnections>
</vertex>
</constraint>

```

A.2 Edge Constraint XML File Template

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<constraint name = "null" scope = "edge" type = "null"
description = "null">
  <edge>
    <type>
      <item>null</item>
    </type>
    <sourceConnection permission = "ignore">
      <vertex>
        <uri>null</uri>
      </vertex>
    </sourceConnection>
    <targetConnection permission = "ignore">
      <vertex>
        <uri>null</uri>
      </vertex>
    </targetConnection>
    <lineStyle>
      <type permission = "ignore">
        <item>null</item>
      </type>
    </lineStyle>
    <lineColour>
      <colour permission = "ignore">
        <item>null</item>
      </colour>
    </lineColour>
    <sourceLabel>
      <text permission = "ignore">
        <item>null</item>
      </text>
    </sourceLabel>
```

```

<midLabel>
  <text permission = "ignore">
    <item>null</item>
  </text>
  <textRE permission = "ignore">
    <item>null</item>
  </textRE>
</midLabel>
<targetLabel>
  <text permission = "ignore">
    <item>null</item>
  </text>
</targetLabel>
<labelColour>
  <colour permission = "ignore">
    <item>null</item>
  </colour>
</labelColour>
<sourceDecoration>
  <type permission = "ignore">
    <item>null</item>
  </type>
</sourceDecoration>
<targetDecoration>
  <type permission = "ignore">
    <item>null</item>
  </type>
</targetDecoration>
</edge>
</constraint>

```

```

<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<constraint name = "" scope = "graph" type = ""
description = "">
  <graph type = "graph">
    <cyclic permission = "">
      <value>null</value>
    </cyclic>
    <inEdge permission = "">

    <identicalInEdgeLabels>null</identicalInEdgeLabels>
    </inEdge>
    <outEdge permission = "">

    <identicalOutEdgeLabels>null</identicalOutEdgeLabels
  >
    </outEdge>
    <vertices permission = "">
      <sUri>null</sUri>
      <eUri>null</eUri>
      <unConnected>null</unConnected>
      <unLabelledVertices>null</unLabelledVertices>
      <identicalVLabels>null</identicalVLabels>

    <unUniqueRepresentation>null</unUniqueRepresentation
  >
    <lowerBoundNumber permission = "" value = "">
      <vertex>
        <OR>
          <uri>null</uri>
        </OR>
        <AND>
          <uri>null</uri>
        </AND>
        <combination>
          <uri>null</uri>
        </combination>
      </vertex>
    </lowerBoundNumber>

```



```

    <upperBoundNumber permission = "" value = "">
      <vertex>
        <OR>
          <uri>null</uri>
        </OR>
        <AND>
          <uri>null</uri>
        </AND>
        <combination>
          <uri>null</uri>
        </combination>
      </vertex>
    </upperBoundNumber>
  </vertices>
  <edges permission = "">
    <uri></uri>
    <unLabelledEdges>null</unLabelledEdges>
    <identicalELabels>null</identicalELabels>
    <lowerBoundNumber permission = "" value = "">
      <edge>
        <OR>
          <uri>null</uri>
        </OR>
        <AND>
          <uri>null</uri>
        </AND>
        <combination>
          <uri>null</uri>
        </combination>
      </edge>
    </lowerBoundNumber>
    <upperBoundNumber permission = "" value = "">
      <edge>
        <OR>
          <uri>null</uri>
        </OR>
        <AND>
          <uri>null</uri>
        </AND>
        <combination>
          <uri>null</uri>
        </combination>
      </edge>
    </upperBoundNumber>
  </edges>
</graph>
</constraint>

```

Appendix B

Diagram

Constraint Lists

B.1 State Transition Diagram Main Constraint List

Start State related constraints:

- At most, one Start State is allowed in the graph (graph constraint, hard).
- At least one Start State must be in the graph (graph constraint, soft).
- Start State must only have outgoing transitions (vertex constraint, hard).
- At least one outgoing transition must exist from a Start State (vertex constrain, soft).
- There must be a path between the Start State and every node in the graph (connected graph) (graph constraint, soft).
- Start state must have a unique visual representation.

End State related constraints:

- At most, three (3) End States are allowed in the graph (graph constraints, hard).
- The graph must contain at least one End State (graph constraint, soft).
- End State must only have incoming transitions (vertex constraint, hard).
- At least one incoming transition must exist for an End State (vertex constraint, soft).
- End State must have a unique visual representation (graph constraint, hard).

Non-terminal State related constraints:

- Each Non-Terminal State must have a label (graph constraint, soft).
- Each Non-Terminal State must have a unique label (graph constraint, hard).
- Each Non-Terminal State must have at least one outgoing transition (vertex constraint, soft).
- Each Non-Terminal State must have at least one incoming transition (vertex constraint, soft).

Transition related constraints:

- Outgoing transitions from Start State must have unique labels (graph constraint, hard).
- Outgoing transitions Non-Terminal States must have unique labels (graph constraint, hard).
- Transition labels must start with “out” (edge constraint, soft).

General Graph constraints:

- The graph must be connected (graph constraint, soft).

B.2 State Transition Diagram Constraint Categories

Category 1: Vertex cardinality related constraints.	<ul style="list-style-type: none"> • At most, one Start State is allowed in the graph (graph constraint, hard). • At least one Start State must be in the graph (graph constraint, soft). • At most, three (3) End States are allowed in the graph (graph constraints, hard). • The graph must contain at least one End State (graph constraint, soft).
Category 2: Edge cardinality related constraints.	<ul style="list-style-type: none"> • Start State must only have outgoing transitions (vertex constraint, hard). • At least one outgoing transition must exist from a Start State (vertex constraint, soft). • End State must only have incoming transitions (vertex constraint, hard). • At least one incoming transition must exist for an End State (vertex constraint, soft). • Each Non-Terminal State must have at least one outgoing transition (vertex constraint, soft). • Each Non-Terminal State must have at least one incoming transition (vertex constraint, soft).
Category 3: Unique visual representation constraints.	<ul style="list-style-type: none"> • Start state must have a unique visual representation. • End State must have a unique visual representation (graph constraint, hard).
Category 4: Vertex label related constraints.	<ul style="list-style-type: none"> • Each Non-Terminal State must have a label (graph constraint, soft). • Each Non-Terminal State must have a unique label (graph constraint, hard).
Category 5: Edge label related constraints.	<ul style="list-style-type: none"> • Outgoing transitions from Start State must have unique labels (graph constraint, hard). • Outgoing transitions Non-Terminal States must have unique labels (graph constraint, hard). • Transition labels must start with “out” (edge constraint, soft).
Category 6: Path constraint.	<ul style="list-style-type: none"> • There must be a path between the Start State and every node in the graph (connected graph) (graph constraint, soft).

B.3 Use Case Diagram Main Constraint List

Use Case Diagram Main Constraint List

Actor related constraints:

- At most (upper bound number) 3 actors can exist in the diagram (graph constraint, hard).
- At least (lower bound number) one Actor must exist in the diagram (graph constraint, soft).
- It is not allowed to connect two Actor vertices with Association, Include, or Extend (edge constraint, hard).
- It is not allowed to connect Actor to Use Case using Include, Extend, or Generalisation edge (edge constraint, hard).
- All Actor vertices must be labelled (graph constraint, soft).
- All Actor labels start with 'capital letter' (vertex constraint, soft).
- All Actor vertices must have labels (graph constraint, soft).
- Actor labels must not be identical (graph constraint, soft).
- Each actor at least should be connected to a Use Case or Another actor (vertex constraint, soft).

Use Case related constraints:

- At most (upper bound number) 3 Use Cases can exist in the diagram (graph constraint, hard).
- At least (lower bound number) one Use Case must exist in the diagram (graph constraint, soft).
- It is not allowed to connect two Use Case vertices with Generalisation or Association Edge (edge constraint, hard).
- It is not allowed to connect Use Case (source) to actor (target) using any edge (edge constraint, hard).
- Every Use Case (as target) at least (lower bound number) should be connected with either an actor or Use Case (vertex constraint, soft).
- All Use Cases must be labelled (graph constraint, soft).
- Use case labels must not be identical (graph constraint, hard).

Edges related constraints:

- All include edges must have <<include>> label (edge constraint, soft).
- All extend edges must have <<extend>> label (edge constraint, soft).

B.4 Use Case Diagram Constraint Categories

Category 1: Vertex cardinality related constraints	<ul style="list-style-type: none"> At most (upper bound number) 3 actors can exist in the diagram (graph constraint, hard). At least (lower bound number) one actor must exist in the diagram (graph constraint, soft). At most (upper bound number) 3 Use Cases can exist in the diagram (graph constraint, hard). At least (lower bound number) one Use Case must exist in the diagram (graph constraint, soft).
Category 2: Edge cardinality related constraints.	<ul style="list-style-type: none"> It is not allowed to connect two Actor vertices with Association, Include, or Extend (edge constraint, hard). It is not allowed to connect Actor to Use Case using Include, Extend, or Generalisation edge (edge constraint, hard). It is not allowed to connect two Use Case vertices with Generalisation or Association edge (edge constraint, hard). It is not allowed to connect Use Case (source) to actor (target) using any edge (edge constraint, hard).
Category 3: Vertex target connection with logical operator.	<ul style="list-style-type: none"> Every Use Case (as target) at least (lower bound number) should be connected with either an Actor or Use Case (vertex constraint, soft).
Category 4: Vertex label related constraints	<ul style="list-style-type: none"> All Actor vertices must be labelled (graph constraint, soft). All Actor labels start with 'capital letter' (vertex constraint, soft). All Actor vertices must have labels (graph constraint, soft). Actor labels must not be identical (graph constraint, soft). All Use Cases must be labelled (graph constraint, soft). Use case labels must not be identical (graph constraint, hard).
Category 5: Edge label related constraints	<ul style="list-style-type: none"> All include edges must have <<include>> label (edge constraint, soft). All extend edges must have <<extend>> label (edge constraint, soft).
Category 6: Vertex source connection with logical operator.	<ul style="list-style-type: none"> Each actor at least should be connected to a Use Case or another Actor (vertex constraint, soft).

Appendix C

Study One

C.1 Study One Constraint Lists

C.1.1 Constraint List for User 1, *State Transition Diagram*

- At most (upper bound), three (3) End States are allowed in the graph (graph, hard).
- Start State must not have incoming transitions (upper bound) (vertex, hard).
- Start state must have a unique visual representation (graph, hard).
- Each Non-Terminal State must have a label (graph, soft).
- Outgoing transitions Non-Terminal States must have unique labels (graph, hard).
- There must be a path between the Start State and every node in the graph (connected graph) (graph, soft).

C.1.2 Constraint List for User 1, *Use Case Diagram*

- At least (lower bound number) one Actor must exist in the diagram (graph, soft).
- It is not allowed to connect two Actor vertices using Association, Include, or Extend edge (edge, hard).
- Every Use Case (as target) at least (lower bound number) should be connected with either an Actor or another Use Case (vertex, soft).
- All Use Case vertices must have labels (graph, soft).
- All Include edges must have “<<include>>” label (edge, soft).
- Each Actor at least (lower bound number) should be connected to a Use Case or another Actor (vertex, soft).

C.2 Study One Post-Experiment Questionnaire

C.3 Study One Exit Questionnaire / Interview

Appendix D

Study Two

D.1 Study Two Questionnaire:

1. Which one of the following expressions is more understandable to you.

- ☐ At most, one Start State is allowed in the graph.
- ☐ It is not allowed to have more than one Start State.
- ☐ None.

2. Which one of the following expressions is more understandable to you.

- ☐ At least one End State must exist in the graph.
- ☐ It is not allowed to have less than one End State in the graph.
- ☐ None.

3. Which one of the following expressions is more understandable to you.

- ☐ Start State must only have outgoing edges.
- ☐ Start State must not have incoming edges.
- ☐ None.

4. Which one of the following expressions is more understandable to you.

- ☐ At least one outgoing edge from a Start State must exist.
- ☐ Edges outgoing from Start State must not be less than 1.
- ☐ None.

5. Which one of the following expressions is more understandable to you.

- ☐ Start State must have unique representation.

- It is not allowed for any other vertex type to share properties with Start State type.
- None.

6. Which one of the following expressions is more understandable to you.

- Each Non-Terminal State must have label.
- It is not allowed to have unlabeled Non-Terminal State.
- None.

7. Which one of the following expressions is more understandable to you.

- Each Non-Terminal State must have unique label.
- It is not allowed to have Non-Terminal States with identical labels.
- None.

8. Which one of the following expressions is more understandable to you.

- Edges outgoing from Start State must have unique labels.
- It is not allowed for edges outgoing from Start State to have identical labels.
- None.

9. Which one of the following expressions is more understandable to you.

- Transition labels must start with the word "out".
- It is not allowed for transition labels to start with anything other than the word "out".
- None.

10. Which one of the following expressions is more understandable to you.

- ☐ There must be a path between the Start State and every node in the graph.
- ☐ It is not allowed to have Start State that has no path to the rest of the vertices in the graph.
- ☐ None.

Appendix E

Study Three

E.1 Study Three Constraint Lists

E.1.1 Constraint List 1

- The diagram cannot have less than one Start State.
- Start State must only have outgoing Transitions.
- In any given diagram, the Start State cannot have exactly the same visual representation as any other vertices in that diagram.
- Each Non-Terminal State must have a label.
- Outgoing transitions from Non-Terminal States cannot have identical labels.
- There must be a path between the Start State and every other State in the diagram.

E.1.2 Constraint List 2

- At most, three (3) End States are allowed in the diagram.
- Non-Terminal State incoming transition must not be less than one.
- For any given diagram, End State must have a unique visual representation in that diagram.
- Non-Terminal States cannot have identical labels.
- Transition labels must start with the substring “out”.
- It is not allowed to have a Start State without a path to every other State in the diagram.

E.2 Questions Per Constraint

E.3 Study Three Post-Experiment Questionnaire

E.4 Study Three Exit Questionnaire / Interview

Appendix F

Study Four

F.1 Inference Features

F.1.1 State Inference Features

- The diagram contains single vertex.
- The diagram contains more than one vertex without edges (not connected diagram).
- The diagram has a single structure (more than one vertex connected together).
- The diagram has multiple structures.
- The properties of a vertex changed to look like another vertex.
- The vertex label has changed.
- The edge label has changed.
- The graph contains all the vertex types available in the diagram.
- All the vertices of the graph are of different types.
- The graph has single source vertex and multiple target vertices.
- The graph has single target vertex and multiple source vertices.
- Vertices in the graph have identical labels.
- Vertices in the graph have different labels.
- Edges in the graph have identical labels.
- Edges in the graph have different labels.

F.1.2 Action Inference Features

- A vertex is deleted from the graph.
- An edge is deleted from the graph.
- Identical part (sub-string) of labels deleted from different vertices.
- At least two properties changed from a vertex.
- A vertex label is deleted.

- An edge label is deleted.

F.1.3 Visual Generalisation Inference Features

- The graph has two structures with the same source vertex type.
- The graph has two structures with the same target vertex type.
- The graph is partially unified (generalised based on the element types in the examples).

F.1.4 Rule Generalisation Features

- The element (vertex or edge) type such as “Start State” and “Actor”.
- Number of vertex source connections (outgoing edges from a vertex).
- Number of vertex target connections (incoming edges to a vertex).
- Vertex type at an edge source connection (the vertex from which the edge is going out).
- Vertex type at an edge target connection (the vertex into which the edge is going).

F.2 Study Four Constraint Lists

F.2.1 Constraint List 1

- At most 3 StartState (s) are allowed in the diagram. (graph, hard)
- NonTerminalState (s) must have unique labels. (graph, hard)
- StartState must have unique visual representation in any given diagram. (graph, hard)
- Any Transition edge label must start with the substring “out”. (edge, soft)
- There should be a path between Red StartState and the Green NonTerminalState in the diagram. (graph, soft)
- There must be at least 1 Red Transition edge connecting StartState (source) with NonTerminalState or EndState (target). (vertex, soft)

F.2.2 Constraint List 2

- At most 4 EndState (s) are allowed in the diagram. (graph, hard)
- NonTerminalState (s) must have labels. (graph, hard)
- EndState must have unique visual representation in any given diagram. (graph, hard)
- Any Transition edge label must start with the substring “_in”. (edge, soft)
- There must be a path between the Yellow NonTerminalState and Blue EndState in the diagram. (graph, soft)
- There must be at least 1 Green Transition edge connecting NonTerminalState (source) with NonTerminalState or EndState (target). (vertex, soft)

F.2.3 Constraint List 3

- It is not allowed to have more than 3 StartState (s) in the diagram. (graph, hard)
- It is not allowed for NonTerminalState (s) to have identical labels. (graph, hard)
- In any given diagram, the StartState cannot have exactly the same visual representation as any other vertices in that diagram. (graph, hard)
- Transition edge labels cannot start with anything other than the substring “out”. (edge, soft)
- It is not allowed to have a Red StartState without a path between it and the Green NonTerminalState in the diagram. (graph, soft)
- It is not allowed to have StartState (source) that is not connected to either a NonTerminalState or an EndState (target) using at least 1 Red Transition edge. (edge or vertex, soft)

F.2.4 Constraint List 4

- It is not allowed to have more than 4 EndState (s) in the diagram. (graph, hard)
- It is not allowed to have unlabelled NonTerminalState (s). (graph, hard)
- In any given diagram, the EndState cannot have the same visual representation as any other vertices in that diagram. (graph, hard)

- Transition edge labels cannot start with anything other than the substring “_in”. (edge, soft)
- It is not allowed to have a Yellow NonTerminalState without a path between it and the Blue EndState in the diagram. (graph, soft)
- It is not allowed to have a NonTerminalState (source) that is not connected to either a NonTerminalState or an EndState (target) using at least 1 Green Transition edge. (vertex, soft)

F.3 Study Four Post-Experiment Questionnaire

F.4 Study Four Exit Questionnaire / Interview

